KOZMA LÁSZLÓ VARGA LÁSZLÓ

A szoftvertechnológia elméleti kérdései

ELTE EÖTVÖS KIADÓ

A szoftvertechnológia elméleti kérdései

Kozma László, Varga László

A szoftvertechnológia elméleti kérdései

A könyv az Oktatási Minisztérium támogatásával
a Felsőoktatási Pályázatok Irodája által lebonyolított
tankönyv-támogatási program keretében jelent meg.

© Kozma László és Varga László, 2003

ISBN 963 463 648 9

Kiadja az ELTE Eötvös Kiadó
Felelős kiadó: Pándi András

TARTALOMJEGYZÉK

ELŐSZÓ 3
SZEKVENCIÁLIS PROGRAMOK 6
BEVEZETÉS 7
SZOFTVERTECHNOLÓGIA 7
OBJEKTUM ELVŰ PROGRAMOZÁS, TÍPUSOSZTÁLY 15
AZ ABSZTRAKT ADATTÍPUS FOGALMA 16
A KÖNYVBEN HASZNÁLT JELÖLÉSEK 18
HALMAZELMÉLETI JELÖLÉSEK ÉS AZOK JELENTÉSE 18
RELÁCIÓK JELÖLÉSE ÉS TULAJDONSÁGAIK 18
FÜGGVÉNYEK 19
AZ ABSZTRAKT ADATTÍPUS EGY MATEMATIKAI MODELLJE 20
Σ-ALGEBRÁK KÖZÖTTI HOMOMORFIZMUS 25
ABSZTRAKT ADATTÍPUSOK ÉS UNIVERZÁLIS ALGEBRÁK 26
SPECIFIKÁCIÓMORFIZMUS 32
ADATTÍPUSOK SPECIFIKÁCIÓJA 37
AZ ADATTÍPUS OSZTÁLYÁNAK SPECIFIKÁCIÓJA 47
A TÍPUSOSZTÁLY FORMALIZÁLT LEÍRÁSA 49
AZ ADATTÍPUSOK OSZTÁLYÁNAK JELLEGZETES SPECIÁLIS ESETEI 52
A TÍPUSOSZTÁLY SPECIFIKÁCIÓJÁNAK ELEMZÉSE 69
AZ ABSZTRAKT SPECIFIKÁCIÓ ELEMZÉSE 69
A KETTŐS SPECIFIKÁCIÓVAL KAPCSOLATOS ALAPFOGALMAK 82
AZ IMPLEMENTÁCIÓ HELYESSÉGE 91
A MEGVALÓSÍTÁSOK ELEMZÉSE 94
A KETTŐS SPECIFIKÁCIÓ HELYESSÉGÉNEK KÉTFÉLE MEGKÖZELÍTÉSE 102
PROCEDURÁLISAN ADOTT KONKRÉT SPECIFIKÁCIÓ ELŐ- ÉS UTÓFELTÉTELEKKEL ADOTT ABSZTRAKT SPECIFIKÁCIÓ SZERINTI HELYESSÉGE 108
SZEKVENCIÁLIS PROGRAMOK 110
A PROGRAMHELYESSÉG FOGALMAI 116
A PROGRAM HELYESSÉGI TULAJDONSÁGAINAK BIZONYÍTÁSA 117
VALÓDI PROGRAMOK PARCIÁLIS HELYESSÉGÉNEK BIZONYÍTÁSA 117
VALÓDI PROGRAMOK TELJES HELYESSÉGÉNEK BIZONYÍTÁSA 119
FLOYD MÓDSZERÉNEK FORMÁLIS DEFINÍCIÓJA 124

Induktív állítások módszere programok parciális helyességének bizonyítására ... 127
A Floyd-féle induktív állítások módszere programok parciális helyességének bizonyítására ... 128
A Floyd-féle induktív állítások módszere programok teljes helyességének bizonyítására ... 128
A Hoare-módszer ... 135
Strukturált programok esetén a Floyd-módszer és a Hoare-módszer ekvivalens ... 146
A bizonyítási rendszer megbízhatósága ... 149
A Hoare-módszer teljességi tétele ... 153
A strukturált programozás alaptétele ... 158
Típusöröklődés ... 165
Specializációval történő típusöröklődés a specifikáció szintjén ... 167
Újrafelhasználással történő típusöröklődés ... 173
Konkurens rendszerek ... 180
A konkurens programozás alapfogalmai ... 182
A párhuzamos programozás problémakörének szemléltető példái ... 188
Diszjunkt determinisztikus programokból álló párhuzamos programok ... 201
Párhuzamos programok közös használatú változókkal ... 211
Párhuzamos programok szinkronizációval ... 227
A kölcsönös kizárás problémája ... 237
A kiéheztetés ... 248
Egy módszer párhuzamos programok előállítására ... 268
A termelő–fogyasztó rendszer ... 275
Absztrakt megvárakoztató utasítások megvalósítása konkrét szemaforutasításokkal ... 281
Osztott rendszerek ... 295
Objektumokhoz való hozzáférés specifikációja a szinkronizáció követelményeinek megfelelően ... 302
Egy absztrakt specifikációs nyelv a szinkronizáció követelményének leírására ... 304
A szinkronizációs specifikáció elemzése ... 317
A holtpontmentesség vizsgálata ... 317
A konzisztencia vizsgálata ... 324
Egy módszer a szinkronizációs specifikáció konkrét megadására ... 329
Az állapottér módszere ... 330
A konkrét specifikáció elemzése ... 338
A holtpontmentesség vizsgálata ... 338
A konzisztencia vizsgálata ... 339
Párhuzamosságot vezérlő modul ... 344
Függelék ... 349
Az UML jelölésrendszere ... 349
Irodalomjegyzék ... 357
Tárgymutató ... 366

ELŐSZÓ

A könyv elsősorban egyetemi hallgatóknak készült. Azoknak az előadásoknak az anyagát tartalmazza, amelyeket a szerzők az Eötvös Loránd Tudományegyetemen a programtervező matematikus hallgatóknak a 8. és a 9. félév során adnak le. Ebben az előadásban a szerzők az objektum elvű szoftvertechnológia módszerbeli összetevőjének elméleti vizsgálatával foglalkoznak.

Az informatika felnőtt korba lépett. Ennek az a lényege, hogy ma már az emberi tevékenység legbonyolultabb feladatainak ellátása számára is képes hatékony szoftvereszközöket gyártani. Ezért a szoftver előállításában egyre nagyobb hangsúlyt kap a szoftver gyártásának technológiája. Az utóbbi években a modellezési nyelvek és az emberi gondolkodásmódhoz közel álló objektum elvű megközelítések uralják a programfejlesztési technológiákat. Ez képezi alapvetően a szerzők vizsgálatának tárgyát is.

A szerzők az objektum elvű programozás módszerét a gyakorlat szempontjai szerinti előadásban tárgyalják.

Modellezési nyelvként az UML diagramjai képezik módszereiknek, vizsgálataiknak gyakorlati oldalról az eszközrendszerét. A könyv mellékletként tartalmazza az UML jelöléseit és azok jelentését.

Az elméleti vizsgálatok a gyakorlati problémáknak és módszereknek az absztrakt modelljeire vonatkoznak. Az objektumot úgy fogják fel, mint egy olyan egységet, amelynek szolgáltatásait két védőgyűrű veszi körül. A belső gyűrű statikus, a külső gyűrű pedig dinamikus szempontból határozza meg az objek-

tum szolgáltatásaihoz való hozzáférés feltételét. A hasonló tulajdonságú objektumok objektumosztályt alkotnak, amely az objektumok absztrakt és konkrét megjelenésében együtt képez egy egységet.

Ez az egység matematikailag definiálható. Ez a matematikai modell lehetővé teszi az absztrakt felület adekvát voltának elemzését, azaz annak kimutatását, hogy az osztály a felhasználó által megkívánt szolgáltatásokat nyújtja. A modell lehetőséget nyújt annak bizonyítására, hogy az absztrakt szolgáltatások konkrét megvalósításai helyesek. Ez a reprezentáció, az implementáció és az öröklődés formális kezelését jelenti. Ily módon az elemzések gyakorlati teendőit formálisan megfogalmazhatjuk, és az elemzések lépéseit a matematika eszközeivel bizonyíthatjuk. Ezzel megadjuk az olvasónak azt a segítséget, amely a gyakorlatban elvégzendő feladatok lényegét alkotja, és gyakorlati módszerek alkalmazásában vezérfonalként szolgálhat.

A könyvben a szerzők az osztálymodellnek az elemzése során a következő kérdések megválaszolására adnak módszereket: hogyan lehet az informálisan specifikált probléma osztályszerkezetét megalkotni; osztott rendszerek esetén hogyan származtathatjuk a szinkronizációs feltételeket?

A. Jó-e az osztály absztrakt leírása?
    1. Az osztály absztrakt leírása önmagában helyes-e?
        - Ellentmondás-mentesség.
        - Teljesség.
        - Holtpontmentesség.
        - Kiéheztetésmentesség.
        - Konzisztencia.
    2. Az absztrakt leírás mindent tartalmaz-e, amit a felhasználó az osztály szolgáltatásaitól elvár?
B. Jó-e az objektumosztály konkrét leírása?

1. A konkrét specifikáció önmagában helyes-e?
   - Ellentmondás-mentesség.
   - Teljesség.
   - Holtpontmentesség.
   - Kiéheztetésmentesség.
2. Jó-e az ábrázolás?
3. Jó-e a megvalósítás (implementáció)?

Az elméleti vizsgálatokat, a módszereket a könyvben számos szemléltető példa ill. gyakorlati feladat egészíti ki.

I. RÉSZ

SZEKVENCIÁLIS PROGRAMOK

BEVEZETÉS

AZ ABSZTRAKT ADATTÍPUS EGY MATEMATIKAI MODELLJE

ADATTÍPUSOK SPECIFIKÁCIÓJA

AZ ADATTÍPUS OSZTÁLYÁNAK SPECIFIKÁCIÓJA

A TÍPUSOSZTÁLY SPECIFIKÁCIÓJÁNAK ELEMZÉSE

SZEKVENCIÁLIS PROGRAMOK HELYESSÉGÉNEK BIZONYÍTÁSA

TÍPUSÖRÖKLŐDÉS

1. BEVEZETÉS

Az anyag tárgyalása során feltételezzük, hogy a könyv olvasója a szoftvertechnológia alapvető fogalmaival már korábban megismerkedett. Ismeri a halmazelmélet és a matematikai logika jelölésrendszerét és annak jelentését. Emlékeztetőül azonban ezeknek a legfontosabb eszköztárát összefoglaljuk.

1.1. SZOFTVERTECHNOLÓGIA

Számítógépen a probléma megoldása szoftvertermékkel történik. Ennek a terméknek az előállítása technológiát igényel. Mint minden gyártási technológia, ez is a következőkből áll:

- gyártási eszközök.
- szabványok, ajánlások.
- módszerek.

Mi most a harmadiknak felsorolt komponenssel, a módszerek kérdéskörével foglalkozunk ebben a tanulmányban. Vizsgálataink elsősorban az absztrakt megoldásokra és a konkrét megoldásokra, valamint azok egymáshoz való viszonyának elemzésére irányulnak.

A szoftvertechnológia célkitűzése megfelelő minőségű szoftvertermék határidőre történő előállítása előre megadott költségen belül. A szoftvertermék minőségének jellemzése sokrétű feladat. A minőség lényege azonban az, hogy a program minden olyan szolgáltatást hibátlanul nyújtani tudjon a felhasználó

számára, amelyeket az elvár tőle. Mivel a program készítője és a program felhasználója rendszerint két különböző személy, a programkészítés folyamatában alapvető hibaforrást jelent az, hogy a program készítője nem érti meg egyértelműen a felhasználót. Nem sikerül elkészítenie egy olyan követelményleírást, amely teljesen és egyértelműen rögzítené a felhasználó elképzelését, kívánságát. A program készítője által elkészültnek vélt program ezért hibákat fog tartalmazni, amelyeket azután javítani, a program által nyújtott szolgáltatásokat módosítani, bővíteni kell. A program kidolgozásának fázisát ezért egy – többnyire időigényes, költséges – további fázis követi, amelyet általában **karbantartásnak** szoktak nevezni.

A program életciklusa során alkalmazott programozási módszertől tehát azt várjuk el, hogy oldja meg a felhasználó által elképzelt problémát, és a program felhasználása során tegye lehetővé a szolgáltatások módosítását, a szolgáltatások listájának bővítését a programtermék hosszú élettartama során. Ennek a követelménynek megfelelően épült a szoftvertechnológia módszertani komponense az **absztrakció**ra és a **modularizáció**ra. 1986-ban írta le B. Liskov és J. Guttag [GUT77] azt a megállapítást, hogy a szoftvertechnológia módszerét két alapelvnek kell meghatároznia, az absztrakciónak és a modularizációnak. Először tisztázzuk ezeket a fogalmakat.

- Az **absztrakció** a programozás adott szintjén a megoldás szempontjából lényegtelen részletek elhanyagolásával felírt megoldás.

  Az absztrakció szintjei:

  Paraméteres absztrakció, amikor a függvény jelentését formális paramétereket tartalmazó kiszámítási szabállyal, algoritmussal adjuk meg.

  Specifikáció szerinti absztrakció, amikor a függvény jelentését egy absztrakt rendszerben, esetenként absztrakt megvalósítástól függetlenül adjuk meg.

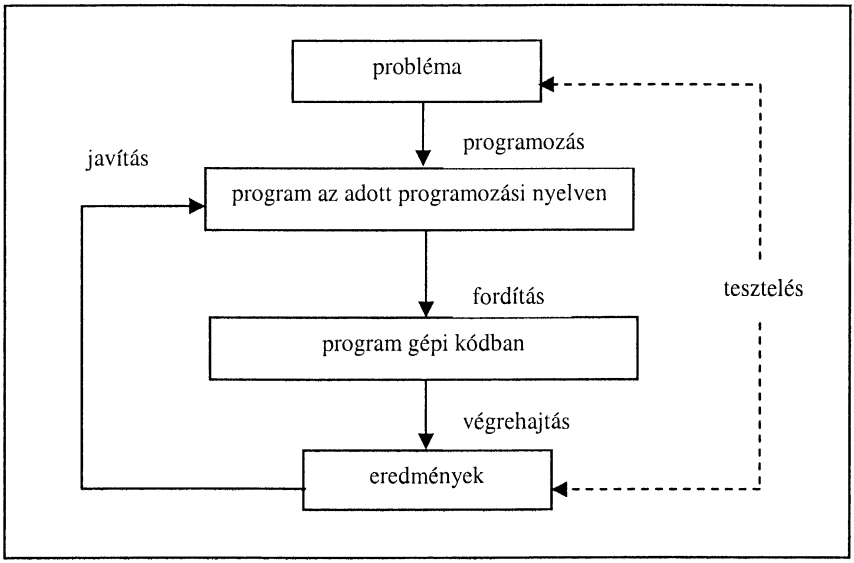
- A **modularizáció** azt jelenti, hogy a probléma megoldását részproblémák megoldására bontjuk. A részprobléma relatív fogalom. A részprobléma megoldását magát is további részproblémák megoldásával adhatjuk meg. A részproblémákra bontást úgy igyekszünk végrehajtani, hogy az egyes részproblémák egymástól független, önálló, de egymással együttműködő programegységekkel legyenek megvalósíthatók, ugyanakkor ezek a részproblémák már jól áttekinthető, hibamentesen megvalósítható programegységeket eredményezzenek. Ezeket a programegységeket nevezzük moduloknak.

  A modularizációnak két formáját szokás alkalmazni:

  - A procedurális elvű megközelítés, amely a részproblémák **eljárásainak moduljait** állítja elő.
  - Az objektum elvű megközelítés, amely viszont **adattípusok osztályait** eredményezi.

A szoftvertechnológia speciális technológia. Eszközei, amelyekkel a terméket előállítja, maguk is programok, amelyek egységes rendszert, ún. szoftverfejlesztési környezetet alkotnak. A termék előállításában rendszerint több szakember vesz részt, akiknek munkáját döntő hányadban nem mechanikus szellemi tevékenység teszi ki. Ez a szellemi munka elsősorban a programfejlesztés különböző fázisainak végrehajtása során használatos módszerek alkalmazását jelenti.
A programok fejlesztésének különböző modelljei ezeket a fázisokat foglalják rendszerbe. A következőkben három programfejlesztési modellt említünk meg azzal a céllal, hogy a könyvünkben tárgyalt problémakört azokban elhelyezhessük. A legegyszerűbb programfejlesztési modellt a következő ábra mutatja.

Egyszerű programfejlesztési modell



Ez általában a kis programok fejlesztésének a modellje. Az eredmények ellenőrzésének egyedüli forrása az, hogy a fejlesztő mennyire értette meg a problémát. A szolgáltatások hibátlansága a tesztelés eredményességétől függ.

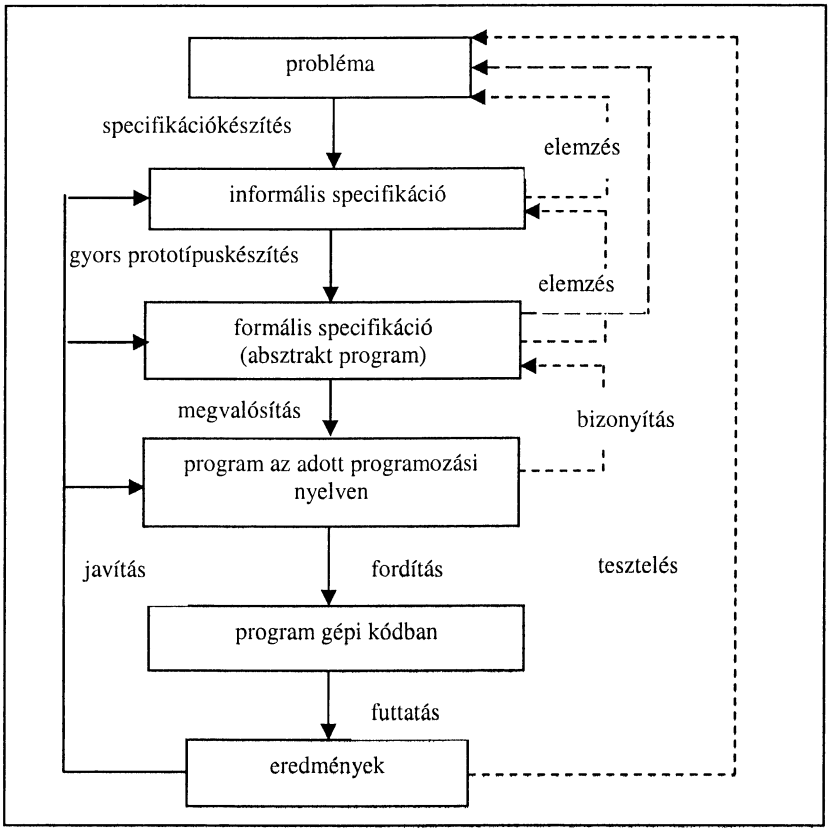
Ezeknek a hátrányoknak a kiküszöbölésére kézenfekvő megoldás lehet a prototípus előállítására épülő programfejlesztési modell.

Ennél a programfejlesztési modellnél először elkészítjük a probléma megoldásának informális specifikációját, amelynek teljességét és helyességét a problémával összevetve elemezni tudjuk.

Ennek alapján készül el egy szintaktikailag és szemantikailag egyértelmű megoldás, egy absztrakt prototípus. Ebben a modellben szerepel egy formális leírás. Ezt szoktuk **specifikáció**nak nevezni. A specifikáció absztrakt leírása a probléma megoldásának, mivel az a konkrét megvalósítás részleteinek az elhanyagolásával adja meg a megoldást. Azt szoktuk mondani, hogy a specifikáció azt írja

le, hogy **mit** kell megoldani és nem azt, hogy hogyan kell azt megoldani. Ugyanakkor a specifikáció mind szintaktikailag, mind szemantikailag egyértelmű.

Programfejlesztési modell absztrakcióval



A formális leírás formálisan elemezhető, analizálható. Az analízisnek az a célja, hogy kimutassa a következőket:

- A specifikáció az informális leírás szerint helyes és vele szemantikailag ekvivalens. (**A specifikáció adekvát**).

- A specifikációban nincsenek egymásnak ellentmondó követelmények. (**Ellentmondás-mentesség**).
- A specifikáció minden, az informális leírásban rögzített problémára megadja a helyes választ. (**Teljesség**).

A specifikáció elemzésére hivatott módszerek ezek kimutatását szolgálják, s könyvünk jelentős részének ez képezi a tárgyát.
A fenti modell esetében a specifikáció által nyújtott megoldás alapján készíthetjük el a program kódját az adott programozási nyelven. A programozási nyelven megadott program – a specifikációhoz hasonlóan – szintén szintaktikailag és szemantikailag egyértelmű megoldása a problémának. A program helyességét a specifikáció szerint igazolni kell. Ez formális úton, bizonyítással történhet. Ezt nevezzük **verifikáció**nak. Ennek végrehajtására szolgálnak a program helyességének bizonyítására kidolgozott módszerek.

A specifikáció különböző módszerei jöttek létre a gyakorlatban. A fent említett elemzések, bizonyítások függnek attól, hogy a specifikáció milyen módszerrel van megadva. A specifikációs módszereket a következőképpen szokták csoportosítani:

- **Állapot elvű specifikáció.** Ezt a módszert szokták nevezni elő- és utófeltételes módszernek és Hoare-féle módszernek is. Formája:

  $$\{\varphi\}\ P\ \{\psi\},$$

  ahol φ és ψ állítások, pl. predikátumkalkulusban felírt formulák, P pedig a specifikálandó programot jelöli. A specifikáció tulajdonképpen a (φ, ψ) kettős. Itt φ az az állítás, amely a program szóba jöhető bemenő adatainak halmazát határozza meg, ψ pedig a bemenő adatok és eredményadatok egymáshoz rendelését definiálja. Például:

  $$\{X \text{ rendezett} \wedge X \neq \{\}\}\ y \leftarrow \max(X)\ \{(\forall x \in X)(y \in X \wedge y \geq x)\}.$$

- **Procedurális specifikáció** (imperatív specifikáció) esetén a definiálandó függvény jelentésének kiszámítási szabályát, a Q eljárást adjuk meg. Itt Q: a függvény értékének kiszámítására szolgáló eljárás.

  Például az előbbi példánknál maradva egy ilyen specifikáció a következő:

  proc-max (X):

  begin if $X = \{\}$ **then** max $\leftarrow \perp$ **else begin let** $x \in X$;

  if $X / \{x\} = \{\}$ **then** max $\leftarrow$ x **else** max $\leftarrow$ largest(X}) **fi end fi end;**

  proc-largest(X):

  begin let $x \in X$; **let** $y \in X/\{x\}$; **if** $x \leq y$ **then** max(X/{x}) **else** max(X/{y}) **fi end;**

- **Axiomatikus specifikáció** (algebrai specifikáció) esetén a függvény jelentését axiómákkal definiáljuk. Legyen A a függvény jelentését meghatározó axiómák halmaza.

  Az előbbi példánk esetében az axiómák halmaza a következő lehet:

$$A = \{X = \{\} \Rightarrow \max(X) = \perp,$$

$$X \neq \{\} \wedge y \in X \wedge X / \{y\} = \{\} \Rightarrow \max(X) = y,$$

$$X \neq \{\} \wedge y \in X \wedge X / \{y\} \neq \{\} \wedge z \in X \Rightarrow \max(X) = \textbf{if } y \geq z$$

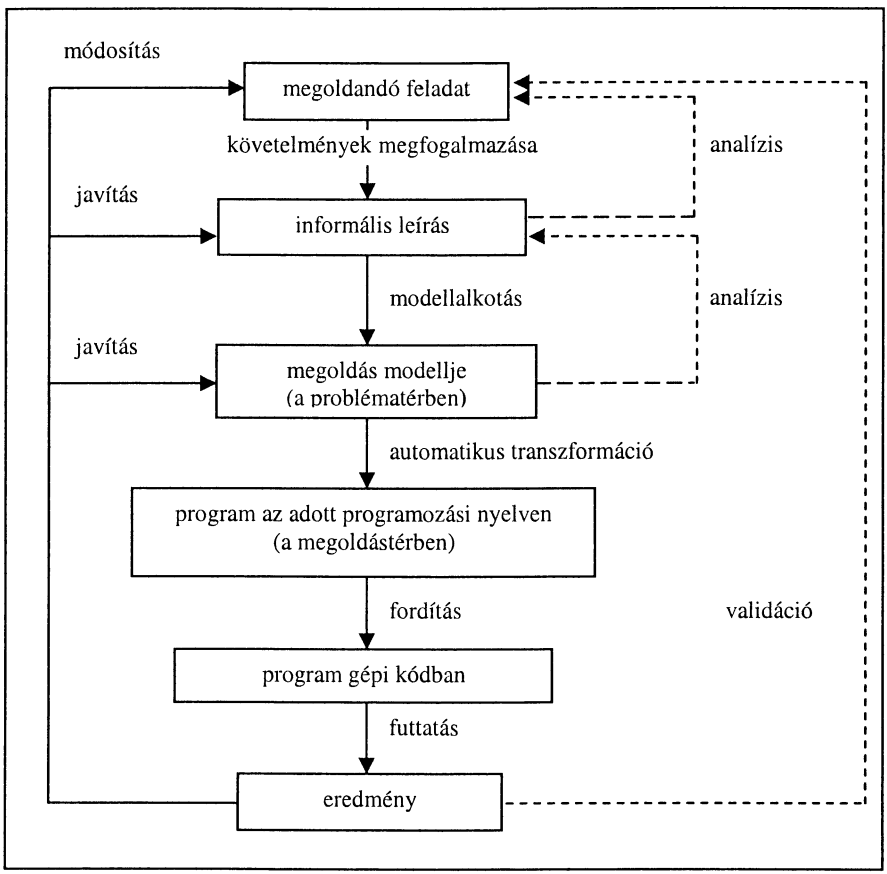
$$\textbf{then } \max(X / \{z\}) \textbf{ else } \max(X / \{y\})\}$$

  Az axiómákat, mint az algebrában, rendszerint predikátumokkal adjuk meg, ezért szokták ezt a specifikációs módszert algebrai specifikációnak is nevezni.

A programfejlesztés egy fejlettebb modelljét mutatja a következő ábránk. Ez a programfejlesztési modell a nagy programrendszerek fejlesztésére kidolgozott modell napjainkban gyakran alkalmazott változata. Ennek a modellnek fontos jellemzője az, hogy a fejlesztés korai szakaszában létrehozzuk a megoldás prototípusát modell formájában. Több modellezési nyelvet is kifejlesztettek erre a

célra. Az egyik ilyen nyelv az UML [UML99], amelyben különböző nézeteknek megfelelő modelleket hozunk létre. A modellezés során az objektum elvűséget követjük. A létrehozott modelleket egyrészt önmagukban vizsgálhatjuk, másrészt a követelményeket tartalmazó specifikációval összevetve elemezhetjük. A helyesnek ítélt modellek alapján azután szoftver eszközök felhasználásával – bizonyos értelemben automatikusan – előállítható a probléma megoldásának programrendszere.

Programfejlesztés modellalkotással



Ez a programfejlesztési modell különösen alkalmas párhuzamos, ill. osztott programrendszerek létrehozására. Ilyen megoldásokban egymástól elkülönülten dolgozó folyamatokról van szó, amelyek időnként egymással üzeneteket váltva, közös erőforrásokat felhasználva oldják meg feladatukat. Ezeknek a folyamatoknak tehát a szinkronizációs követelmények szerint kell helyesen működniük. Könyvünk második részében ennek a problémakörnek a kérdéseivel foglalkozunk.

Kezdetben a program megírása jelentette az adott probléma megoldását. Nagyobb programok esetén ez az út nem volt járható. Általánosan kialakult emberi gyakorlat a probléma megoldásában a modellalkotás. A **programozás** a probléma megoldása a számítástechnikai térben (a megoldástérben). A **modellalkotás** a probléma megoldása valamilyen absztrakt térben. Az **objektum elvű modellalkotás** a probléma megoldását jelenti a valós világ objektumainak terében (a problématérben).

1.2. OBJEKTUM ELVŰ PROGRAMOZÁS, TÍPUSOSZTÁLY

Az objektum elvű programozás lényegét a következőkben foglalhatjuk össze [WEG86]:

Objektum elvűség = adatabsztrakció +
absztrakt adattípus +
típusöröklődés.

ADATABSZTRAKCIÓ Az adat szintaxisának és szemantikájának konkrét megvalósításától független megjelenése.

1.2.1. AZ ABSZTRAKT ADATTÍPUS FOGALMA

DEFINÍCIÓ Az adattípus a típus objektumok halmazának és az objektumokon értelmezett műveletek halmazának az

$$(A, F),$$

kettőse, ahol A a típusobjektumok, F az objektumokon értelmezett műveletek halmaza.

Az **absztrakt adattípus** az adattípusnak egy megvalósítástól független megjelenési formája. Ilyenkor az adattípus definíciója nem teljes definíció. Ez azt jelenti, hogy az adott szinten nem lényeges részletek nélküli definíciót adunk meg. Az absztrakt adattípus valójában „hasonló" tulajdonságokkal rendelkező adattípusok egy osztálya.

A **konkrét adattípus** az absztrakt adattípus megvalósított formája. Az adattípus megvalósítása azt a célt szolgálja, hogy funkcionálisan végrehajtható legyen. Ezt a feladatot a fordítóprogram automatikusan elvégzi azoknak az adattípusoknak az esetében, amelyeket a fordító ismer. Egy korszerű programozási nyelv azonban azt is lehetővé teszi a programozó számára, hogy új, a fordító által nem ismert adattípust definiáljon, megadja annak absztrakt tulajdonságait. A megvalósítás során két feladatot kell megoldanunk:

- Definiálnunk kell azt, hogy az adatokat hogyan kívánjuk a fordítóprogram által ismert más adattípusok felhasználásával ábrázolni (Reprezentáció). Ezek az adattípusok lehetnek olyanok, amelyeket a fordítóprogramba beépítettek a készítők, és olyanok is, amelyeket a program készítője már korábban definiált.
- Definiálnunk kell azt, hogy az új adattípus adatain hogyan kívánjuk elvégezni a műveleteket az ábrázolásnál felhasznált adattípusok műveleteinek felhasználásával. (Implementáció).

A programozó által definiált adattípusok megvalósítása során használt technikák a következők:

1. Átveszünk egy korábban definiált adattípust annak absztrakt, esetleg konkrét tulajdonságaival együtt. Ennek az adattípusnak a tulajdonságait bővítjük, esetleg bizonyos korlátozások között módosítjuk. (Öröklődés).
2. Az adattípust felhasználó számára csak az adattípus deklarációs részét tesszük láthatóvá, a megvalósítási rész láthatatlan marad számára. (Korlátolt láthatóság). A megvalósítási részhez a felhasználó nem tud hozzáférni, nincs tehát lehetősége a deklarációs részben megfogalmazott műveletek funkcionális végrehajtását módosítani, ennek részletei rejtve maradnak számára. (Az információ elrejtése).
3. A felhasználó kicserélheti a megvalósítási részt, de csak úgy, hogy az adattípus látható része, a deklarációs rész, változatlan maradjon. (Az információ beburkolása).
4. A hagyományos programozási nyelvekben a műveletekhez, függvényekhez a fordítás fázisában rendeljük hozzá azok jelentését. (Statikus hozzárendelés). Az objektum elvű programozási nyelvekben a műveletek jelentését, az operációk kiszámítási algoritmusát csak a végrehajtás időpontjában adjuk meg. (Dinamikus hozzárendelés).

A **típusosztály** az adattípus egy komplex definíciója, amely tartalmazza az adattípus absztrakt tulajdonságainak (deklarációs részének) és konkrét tulajdonságainak (megvalósítási részének) a leírását is. Az osztályok önálló programegységek, amelyeket hierarchikus struktúrába és könyvtárba szervezhetünk, újra felhasználhatunk különböző feladatok megoldásához.

TÍPUSÖRÖKLŐDÉS A programfejlesztési technikák alkalmazása során kialakult az a követelmény, hogy a formális leírásokat – a hibák elkerülése érdekében – mechanikusan áttranszformálva újra felhasználhassuk. Adattípus szintjén a tí-

pusöröklődés azt jelenti, hogy az adattípus absztrakt tulajdonságait definiáló specifikáció szintaktikai és szemantikai tulajdonságai mechanikus átírásokkal felhasználhatók a konkrét adattípus specifikációjának elkészítése során. Az öröklődés matematikailag két osztály közötti morfizmust jelent.

1.3. A KÖNYVBEN HASZNÁLT JELÖLÉSEK

1.3.1. HALMAZELMÉLETI JELÖLÉSEK ÉS AZOK JELENTÉSE

$\{\}$: üres halmaz.

$\{a_1, a_2, \ldots, a_n\}$: olyan halmaz, amelynek elemei $a_1, a_2, \ldots, a_n$.

$A \subseteq B$: az A halmaz a B halmaz **részhalmaz**a, másképpen B az A halmaz egy **szuperhalmaza**.

$A \subset B$: $A \subseteq B$ és $A \neq B$.

$A \setminus B$: $\{a \in A \mid a \notin B\}$ **különbséghalmaz**, ha A és B különböző halmazok.

$A_1 \times \ldots \times A_n = \{(a_1, \ldots, a_n) \mid a_i \in A_i,\ 1 \leq i \leq n\}$:

az $A_1, \ldots, A_n$ halmazok Descartes-szorzata.

Speciális halmazok:

$\{0, 1, 2, \ldots\}$ a természetes számok halmaza.

$\{\ldots, -1, 0, 1, 2, \ldots\}$ az egész számok halmaza.

1.3.2. RELÁCIÓK JELÖLÉSE ÉS TULAJDONSÁGAIK

$R \subseteq A \times B$: az R részhalmazt nevezzük relációnak.

A fenti reláció $(a, b) \in R$ elemeinek gyakori felírása a R b.

$R \subseteq A \times A$ lehet:

reflexív, ha $(\forall a \in A)(a\ R\ a)$,

irreflexív, ha $(\sim\exists a \in A)(a\ R\ a)$,

szimmetrikus, ha $a_1, a_2 \in A$ és $a_1 \, R \, a_2 \Rightarrow a_2 \, R \, a_1$,

antiszimmetrikus, ha $a_1, a_2 \in A$ és $a_1 \, R \, a_2$ és $a_2 \, R \, a_1 \Rightarrow a_2 = a_1$,

tranzitív, ha $a_1, a_2, a_3 \in A$ és $a_1 \, R \, a_2$, $a_2 \, R \, a_3 \Rightarrow a_1 \, R \, a_3$,

ekvivalens, ha reflexív, szimmetrikus és tranzitív.

1.3.3. FÜGGVÉNYEK

A függvény, amely A-ról B-re képez le, egy olyan $f \subseteq A \times B$ reláció, amelyre igaz, hogy minden $a \in A$ -hoz létezik legfeljebb egy $b \in B$, amelyre $(a, b) \in f$.

Általános jelölés:

$$f: A \rightarrow B, \; b = f(a).$$

Az f függvény **injektív**, ha minden $b \in B$ esetén **legfeljebb** egy olyan $a \in A$ létezik, amelyre $b = f(a)$.

Az f függvény **surjektív**, ha minden $b \in B$ esetén **legalább** egy olyan $a \in A$ létezik, amelyre $b = f(a)$.

A függvény bijektív, ha injektív és surjektív is egyszerre.

Az f függvényt egy adott A halmazon totális függvénynek nevezzük, ha minden $a \in A$ esetén definiálva van. Az f függvényt egy adott A halmazon parciális függvénynek nevezzük, ha nincs minden $a \in A$ esetén definiálva.

2. AZ ABSZTRAKT ADATTÍPUS EGY MATEMATIKAI MODELLJE

Ebben a részben az adattípus megvalósítástól független tulajdonságait definiáljuk mint speciális univerzális algebrai modellt.

SZIGNATÚRA: $\Sigma = (S, OP)$,

ahol S: szortok véges halmaza.

OP: műveleti szimbólumok véges halmaza.

A műveleti szimbólumok:

$w: s_1 \ldots s_k \rightarrow s$,

ahol w az operáció neve, $s_1, \ldots, s_k, s \in S$, a szortok, ahol $k \geq 0$.

A műveleti szimbólumok között lehetnek konstans operációk is

$OP = K \cup OP_w$,

$k \in K$, $k: \rightarrow S$;

$w \in OP_w$; $w: S^+ \rightarrow S$;

Ha a szignatúra szort része egy halmaznevet tartalmaz, akkor a szignatúrát **homogén** szignatúrának, ha többet, akkor pedig **heterogén** szignatúrának nevezzük.

MEGJEGYZÉS A műveleti szimbólumok a gyakorlatban különböző formákban szoktak megjelenni. Ezek közül mi hármat használunk a formális specifikációkban.

- **Prefix** forma. Például

  add: nat nat → nat. A függvény formája ekkor: k = add(n, m). Ezt fogadjuk el alapértelmezésnek.

- **Infix** forma. Ez az alapértelmezéstől eltérő forma, ezért ezt külön jelöljük az utána elhelyezett "infix" alapszóval. Például

  +: nat nat → nat [infix]. A hozzá tartozó függvényforma: k = n + m.

- **Kifejezés** forma. Ekkor a paraméterek helyét a függvény argumentumában előfordulásuk sorrendjében aláhúzással jelöljük ki a megadott formulában. Például

$$_^{_}: \text{integer} \times \text{integer} \rightarrow \text{integer};$$

  A függvény formája ennek megfelelően $c = b^a$, ahol a, b, c∈integer. Ezt a kifejezés forma egyértelműen azonosítja, ezért nincs szükség külön alapszó használatára.

2.1. PÉLDA Adjuk meg a nat adattípus egy leegyszerűsített formájának szignatúráját.

$$nat = (S_{nat}, OP_{nat}),$$

ahol

$$S_{nat} = \{nat\}, OP_{nat} = \{zerus: \rightarrow nat; suc: nat \rightarrow nat;$$
$$+: nat\ nat \rightarrow nat\ [infix]; \times: nat\ nat \rightarrow nat\ [infix]\}.$$

2.2. PÉLDA Adjuk meg a bool típus szignatúrájának egy leegyszerűsített formáját.

$$bool = (S_{bool}, OP_{bool}),$$

ahol

$$S_{bool} = \{bool\},\ OP_{bool} = \{T: \rightarrow bool;\ F: \rightarrow bool;\ \sim: bool \rightarrow bool,$$

$$\wedge: bool\ bool \rightarrow bool\ [infix]\}.$$

2.3. PÉLDA A Peano-algebra szignatúrája.

$$natbool = (S_{natbool}, OP_{natbool}),$$

ahol

$$S_{natbool} = S_{nat} \cup S_{bool},\ azaz\ S_{natbool} = \{nat, bool\}.$$

$$OP_{natbool} = OP_{nat} \cup OP_{bool} \cup \{\leq: nat\ nat \rightarrow bool\ [infix]\};\ azaz$$

$$OP_{natbool} = \{zerus: \rightarrow nat;\ suc: nat \rightarrow nat;\ +: nat\ nat \rightarrow nat\ [infix];$$

$$\times: nat\ nat \rightarrow nat\ [infix];\ T: \rightarrow bool;\ F: \rightarrow bool;\ \sim: bool \rightarrow bool;$$

$$\wedge: bool\ bool \rightarrow bool\ [infix];\ \leq: nat\ nat \rightarrow bool\ [infix]\}\}.$$

Az $S_{natbool}$ tehát heterogén szignatúra.

A heterogén szignatúra egy más jellegű formájában bizonyos szignatúrák egy másik szort felépítésében vesznek részt. Ezt szemlélteti a következő példa.

2.4. PÉLDA A nem negatív egész értékeket tartalmazó halmaz szignatúrája:

$$natset = (S_{natset}, OP_{natset}),$$

ahol

$$S_{natset} = \{set, natbool\},\ azaz\ S_{natset} = \{set, nat, bool\}$$

$$OP_{natset} = OP_{natbool} \cup \{empty: \rightarrow set,\ insert: set\ nat \rightarrow set,$$

$$delete: set\ nat \rightarrow set,\ has: set\ nat \rightarrow bool\}.$$

DEFINÍCIÓ Adott $\Sigma = (S, OP)$ szignatúrához tartozó **Σ-algebra** egy (S_A, OP_A) kettős, ahol minden $s \in S$ szorthoz hozzárendelünk egy A_s halmazt: $S_A = \{A_s \mid s \in S\}$, és minden $w: s_i\ s_j\ \ldots\ s_k \rightarrow s_l \in OP$ műveleti szimbólumhoz egy

$$w_A: A_{s_i}\ A_{s_j}\ \ldots\ A_{s_k} \rightarrow A_{s_l} \in OP_A$$

operációt, az operáció rövid jelölésével

$$OP_A = \{w_A \mid w \in OP\};$$

Ha ez a leképezés totális függvény, akkor **totális algebráról**, különben pedig **parciális algebráról** beszélünk.

Adott Σ szignatúrához tartozó összes Σ-algebra osztályára használatos jelölés a következő: $Alg(\Sigma)$.

Ha az S szortok halmaza egyelemű, akkor **univerzális homogén algebráról**, ha többelemű, akkor pedig **univerzális heterogén algebráról** beszélünk.

2.5. PÉLDA Egy egyszerű szignatúra és a belőle származtatott Σ-algebrák.

Legyen

$$\Sigma = (\{S\}, \{N_0: \rightarrow S, N_1: S \rightarrow S, N_2: S\ S \rightarrow S\}).$$

Származtassuk ebből a következőket:

1. Legyen S: nat; N_0: zerus; N_1: suc; N_2: add;

   Ekkor a fenti szignatúrából származtatott egyik Σ-algebra:

   $$\Sigma\text{-alg}_1 = (\{nat\}, \{zerus: \rightarrow nat, suc: nat \rightarrow nat, add: nat\ nat \rightarrow nat\}).$$

2. Legyen most

   $$S: N; N_0: 0; N_1: _ + 1; N_2: _ + _ ;$$

   Akkor, a fenti szignatúrából származtatott másik Σ-algebra:

   $$\Sigma\text{-alg}_2 = (\{N\}, \{0: \rightarrow N, _ + 1: N: \rightarrow N, _ + _: N\ N \rightarrow N\}).$$

Adott Σ szignatúrából egymástól lényegesen különböző Σ-algebrák is származtathatók. Erre mutatunk most példát.

2.6. PÉLDA Legyen adott a következő szignatúra:

$$\Sigma = (\{S\}, \{N_0: \rightarrow S, N_1: S \rightarrow S\}).$$

Első származtatás:

$$\Sigma\text{-natalgebra} = (\{\text{nat}\}, \{\text{zerus}: \rightarrow \text{nat}, \text{suc}: \text{nat} \rightarrow \text{nat}\}),$$

(ahol a "succ" operáció jelentése: $(\forall n \in \text{nat})\ (\text{succ}(n) = n + 1)$.

Második származtatás:

$$\Sigma\text{-boolalgebra} = (\{\text{bool}\}, \{\text{"false"}: \rightarrow \text{bool}, \text{suc}: \text{bool} \rightarrow \text{bool}\}),$$

(ahol a "succ" operáció jelentése:

$(\forall n \in \text{bool})$ (succ(n) = ha n = "false" akkor "true", különben "false"))

Eddig csak az algebrák formális, szintaktikai tulajdonságaival foglalkoztunk. A következőkben a szintaktika mellett az algebra jelentésének a megadásával és a teljes specifikációjával foglalkozunk.

SPECIFIKÁCIÓ SPEC = (Σ, E), ahol E a szemantikát meghatározó specifikáció, ami a műveletek kiszámítási szabályainak halmaza, vagy elő- és utófeltételeinek halmaza, vagy axiómák halmaza.

2.7. PÉLDA Tekintsük a 2.4. példánkat és egészítsük ki azt a szignatúra szemantikai részével is.

E: $a, b \in S;\ N_2(a, N_0) = a,\ N_2(a, b) = N_2(b, a),\ N_2(N_1(a), b) = N_1(N_2(a, b))$,

A teljes specifikáció:

$$\text{SPEC} = (\{S\}, \{N_0: \rightarrow S,\ N_1: S \rightarrow S,\ N_2: S\ S \rightarrow S\},$$

$\{a, b \in S;\ N_2(a, N_0) = a,\ N_2(a, b) = N_2(b, a),\ N_2(N_1(a), b) = N_1(N_2(a, b))\})$.

Ennek megfelelően a fenti két származtatott algebra specifikációja:

$\Sigma\text{-alg}_1$ esetében:

$$\text{SPEC}_1 = (\{\text{nat}\}, \{\text{zerus}: \rightarrow \text{nat}, \text{suc}: \text{nat} \rightarrow \text{nat}, \text{add}: \text{nat nat} \rightarrow \text{nat}\},$$

$$n, m \in \text{nat};\ \text{add}(n, \text{zerus}) = n,\ \text{add}(n, m) = \text{add}(m, n),$$

$$\text{add}(\text{succ}(n), m) = \text{succ}(\text{add}(n, m))\}).$$

Σ-alg$_2$ esetében:

$$SPEC_2 = (\{N\}, \{0: \rightarrow N, _+1: N: \rightarrow N, _+_: N\,N \rightarrow N\},$$

$$\{i, j \in N, i + 0 = i, i + j = j + i, (i + 1) + j = (i + j) + 1\}).$$

2.1. Σ-ALGEBRÁK KÖZÖTTI HOMOMORFIZMUS

DEFINÍCIÓ A Σ-algebrák közötti homomorfizmus az alaphalmazok és az operációk között fennálló egymással összefüggő reláció. Legyen adott $\Sigma = (S, OP)$, és legyenek megadva a belőle származtatott $A = (S_A, OP_A)$, $B = (S_B, OP_B)$ Σ-algebrák.

A Σ- homomorfizmus

$$h: A \rightarrow B,$$

egy függvénycsalád

$$h = (h_s),\ s \in S,$$

ahol

$$h_s: S_A \rightarrow S_B$$

úgy, hogy bármely $w \in OP$:

$$w = n: s_1 \ldots s_k \rightarrow s_l,$$

akkor teljesül a homomorfikus feltétel, vagyis:

$$h_s(w_A(a_1 \ldots a_k)) = w_B(h_{s_1}(a_1) \ldots h_{s_k}(a_k)),$$

teljesül minden

$$(a_1 \ldots a_k) \in A_{s_1} \times \ldots \times A_{s_k} \text{ esetén.}$$

A bijektív homomorfizmust **izomorfizmus**nak nevezzük.

Az A és a B Σ-algebrákat izomorfikusnak nevezzük, ha A-ból B-be létezik egy Σ-izomorfizmus. Az izomorfikusság jelölése: $A \cong B$.

DEFINÍCIÓ Egy adott Σ szignatúrához tartozó **absztrakt adattípus** a Σ-algebrák egy olyan osztálya, amely az izomorfizmusra zárt:

$$C \subseteq Alg(\Sigma) \textbf{ és } A \in C \textbf{ és } A \cong B \Rightarrow B \in C,$$

bármely (A, B) Σ-algebrapárra.

Itt tehát a Σ-algebrák hasonlóságáról van szó, olyanokról, amelyek nem különböztethetők meg logikailag, csak formálisan egymástól. Az absztrakt adattípus ezeknek a közös tulajdonságát jeleníti meg.

2.1.1. PÉLDA Tekintsük a következő szignatúrát:

$$\Sigma = (\{s_1, s_2\}, \{N_0: \rightarrow S, N_1: \rightarrow S, N_2: S \rightarrow S, N_3: S\ S \rightarrow S\})$$

Itt $S = \{s_1, s_2\}$. Ebből származtatassunk két Σ-algebrát. Jelöljük ezeket A-val és B-vel:

$$A = (bool, \{T: \rightarrow bool, F: \rightarrow bool, \sim: bool \rightarrow bool,$$
$$\wedge: bool\ bool \rightarrow bool\});$$

$$B = (bin, \{1: \rightarrow bin, 0: \rightarrow bin, vált: bin \rightarrow bin, _ \times _: bin\ bin \rightarrow bin\}).$$

Itt bin = {0, 1} és vált(0) = 1, vált(1) = 0;

Most a h: A $\rightarrow$ B Σ-morfizmus bijektív homomorfizmus, azaz izomorfizmus.

2.2. ABSZTRAKT ADATTÍPUSOK ÉS UNIVERZÁLIS ALGEBRÁK

A homogén univerzális algebra jól ismert példái:

félcsoport = $(A, \{*_A\})$;

monoid = $(A, \{*_A, e\})$;

csoport = $(A, \{*_A, e, (_)^{-1}\})$;

gyűrű = $(A, \{*_A, Z, +, -\})$;

2.2.1. PÉLDA Egy jól ismert félcsoport a következő:

A bázishalmaz neve: A

A műveleti szimbólum: $*$: A A $\rightarrow$ A

A művelet jelentése: a, b, c$\in$A: (a $*$ b) $*$c = a $*$ (b $*$ c)

Az univerzális algebrai vizsgálatoknál is előfordulnak összetettebb szerkezetek. Ezek áttekinthető megadására vezették be a következő jelölésrendszert:

<algebra neve> =

sorts: <bázishalmazok nevei>

oprs: <műveletek nevei és azok formája>

eqns: <a műveletek jelentését egyenletek formájában felíró axiómák>

A félcsoport leírása tehát ebben a formában:

semigroup =

sorts: A

oprs: $*$: A A $\rightarrow$ A

eqns: a, b, c$\in$A;

(a $*$ b) $*$ c = a $*$ (b $*$ c)

A monoid specifikációja:

monoid =

sorts: A

oprs: $*$: A A $\rightarrow$ A

e: $\rightarrow$ A

eqns: a, b, c$\in$A; d$\in$A;

(a $*$ b) $*$ c = a $*$ (b $*$ c)

d $*$ e = e $*$ d

$$d * e = d$$

Ez a jelölésforma egyszerű módon lehetővé teszi az öröklődési technika alkalmazását is a következőképpen:

A monoid specifikációja öröklődéssel.

monoid = semigroup +

oprs: e: $\rightarrow$ A

eqns: d$\in$A;

$$d * e = e * d$$

$$d * e = d$$

Ennek a jelölésnek a jelentése a következő: Másolódjon bele a monoid specifikációjába a semigroup specifikációja rendre az alapszavakhoz.

2.2.2. PÉLDA A csoport specifikációja öröklődéssel.

group = monoid +

oprs: $(_)^{-1}$**:** A $\rightarrow$ A

eqns: m$\in$A;

$$m * (m)^{-1} = e$$

$$(m)^{-1} * m = e$$

DEFINÍCIÓ Az absztrakt adattípus definíciója. Az adattípus (S_A, OP_A) egy speciális Σ-algebra, amelynek speciális tulajdonságai a következők:

1. **Egyszerű adattípus homogén univerzális algebra**

   $S_A = \{A\}$, ahol A a típusobjektumok halmaza. Például nat, bool.

   $OP_A = \{f_0, f_1, \ldots, f_m\}$ parciális leképezések.

   Van legalább egy olyan operáció, amely **konstans operáció**. Jelölje ezt f_0: $\rightarrow$ A. Az A halmaz **megkonstruálható**.

2. Összetett adattípus heterogén univerzális algebra

$S_A = \{A_0, A_1, \ldots A_n\}$, $n \geq 1$, ahol A_0 **kitüntetett** alaphalmaz, amely az összetett adattípus objektumait azonosítja. A többi $A_1, \ldots, A_n$ halmaz elemei, objektumai a típusobjektumok felépítésében, jellemzésében játszanak szerepet.

$OP_A = \{f_0, f_1, \ldots, f_m\}$parciális leképezések.

Van legalább egy olyan **konstans operáció**, amely az A_0 halmaz elemét állítja elő: $f_0: \rightarrow A_0$. Az A_0 halmaz **megkonstruálható**.

DEFINÍCIÓ Az A halmazt akkor nevezzük megkonstruálhatónak, ha az A halmaz minden eleme előállítható az OP_A műveleteinek véges számú egymás utáni alkalmazásával.

Az OP_A műveletei közül azt a legkevesebb számú műveletet, amelyeknek egymás utáni alkalmazásával az S_A halmaz minden eleme előállítható, **konstrukciós műveleteknek**, vagy röviden **konstruktoroknak** nevezzük. A konstruktorokon kívüli operációkat **szelektoroknak**, vagy nem konstruktorukciós műveleteknek nevezzük.

MEGJEGYZÉS Ha a műveleteknek több olyan csoportja is van, amelyek tagjai eleget tesznek a konstrukciós műveletek kritériumának, akkor válasszunk ki közülük egyet és ennek a tagjait nevezzük konstrukciós műveleteknek.

2.2.3. PÉLDA A nem negatív egész számok egy műveletekben gazdagabb specifikációja.

nat =

sorts: nat

oprs: zerus: $\rightarrow$ nat

succ: nat $\rightarrow$nat

add: nat nat → nat

prec: nat → nat

eqns: n, m∈nat;

add(n, m) = add(m, n)

add(n, zerus) = n

add(n, succ(m)) = succ(add(n, m))

prec(zerus) = "undefined"

prec(succ(n)) = n

MEGJEGYZÉS A kivétel (exception) kezelésének problémája. Kérdés, hogyan adjuk meg, mi a prec(zerus) értéke. Például gondolkodhatunk úgy, hogy ez egy speciális érték. Legyen az ⊥, amely "nat"-nak olyan értéke, amelyet sem nem követ, sem nem előz meg valódi érték:

$$succ(\bot) = \bot; \; pred(\bot) = \bot;$$

Igen ám, de ha megköveteljük, hogy a nat típus tulajdonsága kell, hogy legyen minden n esetén a következő:

$$succ(prec(n)) = n,$$

akkor a zerus esetén ez nem áll fenn:

$$succ(prec(zerus)) = succ(\bot) = \bot.$$

Ez a megoldás tehát a programozók számára általában nem jó. Ezért írtuk, hogy a prec művelet nincs értelmezve a zerus értékre:

prec(zerus) = "undefined"

2.2.4. PÉLDA A Boole-algebra szokásos specifikációja.

bool =

sorts: bool

oprs: "true": → bool

"false": → bool

$\sim$: bool $\rightarrow$ bool

$\wedge$: bool bool $\rightarrow$bool [infix]

$\vee$: bool bool $\rightarrow$ bool [infix]

$\Rightarrow$: bool $\Rightarrow$ bool [infix]

eqns: a, b$\in$ bool;

$\sim$"true" = "false"

$\sim$"false" = "true"

a $\wedge$ b = b $\wedge$ a

a $\vee$ b = b $\vee$ a

a $\wedge$ "true" = a

a $\wedge$ "false" = "false"

a $\vee$ "true" = "true"

a $\vee$ "false" = a

a $\Rightarrow$ b = $\sim$ a $\vee$ b

2.2.5. PÉLDA A nem negatív egész számok típusának specifikációja relációkkal.

natbool = nat + bool +

sorts: natbool = nat

oprs: eq: natbool natbool $\rightarrow$ bool

lesseq: natbool natbool $\rightarrow$ bool

eqns: a, b, c$\in$natbool;

eq(a, b) = eq(b, a)

eq(a, a) = "true"

eq(a, b) $\wedge$ eq(b, c) $\Rightarrow$ eq(a, c)

lesseq(zerus, a) = "true"

lesseq(a, b) $\wedge$ lesseq(b, c) $\Rightarrow$ lesseq(a, c)

$$\text{lesseq}(a, b) \wedge \text{lesseq}(b, a) \Rightarrow \text{eq}(a, b)$$

Az adattípusok specifikációjában is szükségünk lehet formális paraméterre. Erre vezetünk be jelölést és szemléltetjük a következő példával.

2.2.6. PÉLDA A nem negatív egész számok specifikációja modulo(m) szerinti műveletekkel.

modnat(m: natbool) = **natbool** +
sorts: modnat = natbool
oprs: reset: natbool → natbool
eqns: n∈natbool;
reset(zerus) = zerus
reset(succ(n)) = **if** n < prec(m) **then** succ(reset(n))
else reset(zerus) **fi**;
end modnat;

Könyvünkben a modulo(n) szerinti összeadást és kivonást rendre a következőképpen jelöljük:

$$\oplus ; \ominus$$

2.3. SPECIFIKÁCIÓMORFIZMUS

Eddigi példáink is mutatták, hogy amikor egy szignatúrát definiálunk, akkor abban gyakran felhasználjuk a már korában definiált szignatúrákat. Erre a módszerre a következőkben még általánosabb formában is szükségünk lesz, ennek definiálásához vezetjük be először a szignatúramorfizmust, majd a specifikációmorfizmus fogalmát.

DEFINÍCIÓ A szignatúramorfizmus definíciója. Legyenek adva a $\Sigma = (S, OP)$ és a $\Sigma' = (S', OP')$ szignatúrák. A szignatúramorfizmus egy $\mu = (\mu_s, \mu_{op})$ $(\mu: \Sigma \rightarrow \Sigma')$ függvénypár úgy, hogy

$$\mu_s: S \rightarrow S';$$

$$\mu_{op}: OP \rightarrow OP';$$

és minden n: $s_1 \ldots s_k \rightarrow s \in OP$, $k \geq 0$ esetén létezik egy olyan m operációnév, amelyre

$$\mu_{op}(n: s_1 \ldots s_k \rightarrow s) = (m: \mu_s(s_1) \ldots \mu_s(s_k) \rightarrow \mu_s(s)).$$

Speciális szignatúramorfizmusok:

- A szignatúra szortjainak és operációinak **átnevezése.**
- A szignatúra **bővítése** új szortokkal és új operációkkal, azaz korábbi szignatúra átvételével történő új szignatúradefiníció.
- A szignatúra új **ábrázolása,** azaz a szignatúra átvétele, és az abban szereplő szortok helyére más szortokból képezett ábrázolások beépítése.

2.3.1. PÉLDA Példa átnevezéssel történő szignatúramorfizmusra. Legyen

$$\Sigma = (\{N\}, \{0: \rightarrow N; _+1: N \rightarrow N; _+_: N\ N \rightarrow N\}),$$

$$\Sigma' = (\{nat\}, \{zerus: \rightarrow nat; succ: nat \rightarrow nat; add: nat\ nat \rightarrow nat\}).$$

Ekkor a $\mu = \Sigma \rightarrow \Sigma'$ morfizmus egy átnevezés, ahol tehát

$$\mu_s(N) = nat,$$

$$\mu_{op}(0: \rightarrow N) = zerus: \rightarrow nat;\ \mu_{op}(_+1: N \rightarrow N) = succ: nat \rightarrow nat;$$

$$\mu_{op}(_+_: N\ N \rightarrow N) = add: nat\ nat \rightarrow nat.$$

2.3.2. PÉLDA Legyen

$$\Sigma = (\{elem_1, elem_2, pár\},$$

$$\{párosít: elem_1\ elem_2 \rightarrow pár;\ első: pár \rightarrow elem_1;\ hátsó: pár \rightarrow elem_2\}),$$

$$\Sigma' = (\{\text{nat, nat, natpár}\},$$

$$\{\text{összerak: nat nat} \rightarrow \text{natpár; fej: natpár} \rightarrow \text{nat;}$$

$$\text{törzs: natpár} \rightarrow \text{nat }\}).$$

Itt morfizmusok a következők:

$$\mu_s(\text{elem}_1) = \mu_s(\text{elem}_2) = \text{nat};\ \mu_s(\text{pár}) = \text{natpár};$$

$$\mu_{op}(\text{párosít: elem}_1\ \text{elem}_2 \rightarrow \text{pár}) = \text{összerak: nat nat} \rightarrow \text{natpár};$$

$$\mu_{op}(\text{első: pár} \rightarrow \text{elem}_1) = \text{fej: natpár} \rightarrow \text{nat};$$

$$\mu_{op}(\text{hátsó: pár} \rightarrow \text{elem}_2) = \text{törzs: natpár} \rightarrow \text{nat};$$

amelyek szintén átnevezések.

2.3.3. PÉLDA Ábrázolással létrejövő morfizmus.

$$\Sigma = (\{\text{stack, elem}\},$$

$$\{\text{create:} \rightarrow \text{stack; push: stack elem} \rightarrow \text{stack; pop: stack} \rightarrow \text{stack elem}\}),$$

$$\Sigma' = (\{\text{vector, nat, elem}\}, \{\text{create}_c\text{:} \rightarrow \text{vector nat;}$$

$$\text{push}_c\text{: vector nat elem} \rightarrow \text{vector nat; pop}_c\text{: vector nat} \rightarrow \text{vector nat elem}\})$$

Most

$$\mu_s(\text{stack}) = \text{vector nat};\ \mu_s(\text{elem}) = \text{elem};$$

$$\mu_{op}(\text{create:} \rightarrow \text{stack}) = \text{create}_c\text{:} \rightarrow \text{vector nat;}$$

$$\mu_{op}(\text{push: stack elem} \rightarrow \text{stack}) = \text{push}_c\text{: stack} \rightarrow \text{stack elem;}$$

$$\mu_{op}(\text{pop: stack} \rightarrow \text{stack elem}) = \text{pop}_c\text{: vector nat} \rightarrow \text{vector nat elem;}$$

Kitüntetett elemű szignatúramorfizmus, ha

$$\mu_s(\text{pt}(\Sigma)) = \text{pt}(\Sigma').$$

A verem esetében kitüntetett elemű szignatúramorfizmusról van szó, mivel

$$\text{pt}(\Sigma) = \text{stack};\ \text{pt}(\Sigma') = \text{vector nat;}$$

így

$$\mu_s(\text{stack}) = \text{vector nat.}$$

A bennfoglaltatás a bővítés egyszerű esete: $\Sigma \subseteq \Sigma'$.

DEFINÍCIÓ A szignatúramorfizmus kiterjesztése változókra. Legyenek megadva a $\Sigma = (S, OP)$ és $\Sigma' = (S', OP')$ szignatúrák és a $\mu: \Sigma \to \Sigma'$ szignatúramorfizmus. Jelölje X és X' rendre a Σ ill. Σ' szignatúrákban szereplő változók egy halmazát. A

$$\mu_X: (\cup X_s)_{\forall s \in S} \to (\cup X'_{s'})_{\forall s' \in S'}$$

injektív függvényt a μ szignatúramorfizmus változóira történő kiterjesztésének nevezzük, ha valahányszor

$$x \in X_s,\ s \in S,$$

akkor

$$\mu_X(x) \in X'_{\mu}(s).$$

DEFINÍCIÓ A szignatúramorfizmus kiterjesztése "term"-ekre. Minden $t \in T_{\Sigma(X)}$ "term"-re, bármely $x \in X$ esetén.

$$\mu_T(x) = \mu(x);$$

bármely $n: \to s \in OP$ esetén, amelyre $\mu(n: \to s) = (m: \to \mu(s))$,

$$\mu_T(n) = m;$$

bármely $n: s_1,\ldots,s_k \to s \in OP$ esetén, amelyre

$$\mu(n: s_1,\ldots,s_k \to s) = (m: \mu(s_1),\ldots,\mu(s_k) \to \mu(s));$$

$$\mu_T(n(t_1, \ldots, t_k)) = m(\mu_T(t_1), \ldots, \mu_T(t_k)).$$

2.3.4. PÉLDA Legyen

$$\Sigma = (\{N\}, \{0: \to N;\ _+1: N \to N;\ _+_: N\ N \to N\},$$
$$\{a, b \in N;\ a + b = b + a,\ a + 0 = a,\ a + (b + 1) = ((a + b) + 1)\});$$

$$\Sigma' = (\{nat\}, \{zerus: \rightarrow nat; succ: nat \rightarrow nat; add: nat\ nat \rightarrow nat\},$$

$$\{a, b \in nat; add(a,b) = add(b, a), add(a, 0) = a,$$

$$add(a, succ(b)) = succ((add(a, b))\});$$

akkor például:

$$\mu_T(a + (b + 1)) = add(a, succ(b));$$

DEFINÍCIÓ A specifikációmorfizmus definíciója. A h: SPEC $\rightarrow$ SPEC', ahol $h = (\mu_s, \mu_{op}, \mu_E)$; specifikációmorfizmus a szemantikát meghatározó E-ben szereplő változókra és "term"-ekre kiterjesztett szignatúramorfizmus úgy, hogy

$$\mu(E) \subseteq E'.$$

A következőkben a specifikációmorfizmust adattípusok, adattípus osztályok specifikációjának tömörebb formában való felírására és az öröklődés definiálására használjuk fel.

3. ADATTÍPUSOK SPECIFIKÁCIÓJA

DEFINÍCIÓ Az adattípus specifikációja egy

(PAR, e, EXP)

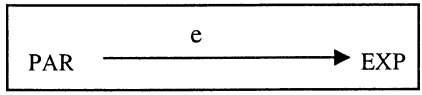
hármas, ahol

PAR: specifikáció, a paraméterek tulajdonságának leírása.

EXP: kitüntetett szortú specifikáció, a típusobjektumok halmazának és a rajta értelmezett műveletek szintaxisának és szemantikájának a definíciója.

e: specifikáció morfizmus, amely a paraméterek specifikációját az "exp" specifikációra képezi le. A mi esetünkben ez a morfizmus tartalmazást jelent, ami természetesen magába foglalja a bővítés lehetőségét is.

Az adattípus specifikációjának morfizmusdiagramja a következő:



Az összetett adattípusok specifikációjára célszerű, ennek a definíciónak megfelelően, a következő formát használni:

⟨a típus neve⟩(⟨a paraméterek listája⟩) **is a type specification =**

parameters = <az átvett aktuális típusnév$_1$> + ...

+ <az átvett aktuális típusnév$_k$> +

sorts: <a formális paraméterek nevei>;

oprs: <a műveletek formái>;

eqns: <a műveletek jelentésének leírása>;

exports =

type sorts: <a típushalmaz neve>;

oprs: <a műveletek formái>;

eqns: <a műveletek jelentésének leírása>;

end ⟨a típus neve⟩;

Itt a paraméterrészben egyszerű öröklődéssel átvehetünk már ismert specifikációkat azonosítójuk felsorolásával.

A specifikációtól elvárjuk, hogy teljes és ellentmondásmentes legyen. A specifikáció ezen tulajdonságainak formális bizonyítása általában bonyolult feladat. Az adattípus specifikációjánál azonban ezt a két tulajdonságot a következő módon érhetjük el.

A konstrukciós műveletek szemantikája az, hogy azoknak egymás utáni alkalmazásával a szóban forgó típusobjektumok halmazának minden eleme előállítható. A nem konstrukciós műveletek jelentését viszont adjuk meg minden egyes típusobjektumra. Ezt tehát úgy tehetjük meg, hogy minden nem konstrukciós műveletnek megadjuk a jelentését minden konstrukciós művelet által előállított értékre:

$$f_s(f_c(a)) = \ldots$$

ahol f_s nem konstrukciós műveletet, f_c pedig konstrukciós műveletet jelöl.

Célszerű ugyanazt a nevet használni a típusobjektumok halmazára (type sort), mint magának az adattípusnak a megnevezésére.

3.1. PÉLDA Legyenek a natboolx adattípus műveletei a következők:

Konstrukciós műveletek:

0: a nulla érték előállítása.

n + 1: az n-re rákövetkező egész szám előállítása.

Nem konstrukciós műveletek:

add(n + m): a két nem negatív egész szám összeadása.

square(n): az n szám négyzetének előállítása.

even(n): ha n páros, akkor igaz értéket ad, különben az érték hamis.

if b **then** n **else** m **fi**; ha b igaz, akkor n, különben m.

Készítsük el a tanult specifikációs formában a natboolx típus teljes és ellentmondásmentes specifikációját, öröklődéssel átvéve a **bool** típus specifikációját.

MEGOLDÁS

natboolx **is a type specification** =

parameters: bool +

exports:

type sorts: natboolx

oprs : 0: → natboolx

_ +1 : natboolx → natboolx

add: natboolx natboolx → natboolx

square: natboolx natboolx → natboolx

even: natboolx → bool

if_then_els _fi: bool natboolx natboolx → bool

eqns: n, m∈natboolx;

add(n, m) = add(m, n)

add(n, 0) = n

add(n, (m + 1)) = add(n, m) + 1

even(0) = "true"

even(m + 1) = ~even(m)

square(0) = 0 + 1

square(n + 1) = add(add(square(n), n), n) + 1

if "true" **then** n **else** m **fi** = n

if "false" **then** n **else** m **fi** = m

end natboolx;

3.2. PÉLDA Adjuk meg a vektor típus specifikációját a megismert formalizmussal. A vektor műveletei legyenek a következők:

Konstrukciós műveletek:

nil: üres vektor létrehozása.

A[j]::= e: értékadás az "A" vektor j-edik elemének.

Szelekciós művelet:

A[j]: a j-edik elem értékének kiolvasása az "A" vektorból.

MEGOLDÁS

vector(natbool, elem) **is a type specification** =

parameters: natbool +

sorts: elem

oprs: ⊥: → elem

exports:

type sorts: vector

oprs: ⊥: → elem

nil: → vector

_ [_]::= _ : vector natbool elem → vector

_ [_]: vector natbool → elem

eqns: v ∈ vector; i, j∈natbool; e∈elem;

nil[i] = ⊥

(v[i]::= e)[j] = **if** i = j **then** e **else** v[j]

end vector;

3.3. PÉLDA Adva van a verem következő informális specifikációja.

Konstrukciós műveletek:

create: üres sor előállítása.

push(s, a): egy a elem elhelyezése az s sor végén.

Nem konstrukciós műveletek:

rem(s): az első elemétől megfosztott sor előállítása s-ből.

head(s): az s sorban első helyen álló elem előállítása.

length(s): az s sorban lévő elemek számát adja meg eredményül.

A veremben lévő elemek száma nem lehet nagyobb, mint n, $n > 0$. Készítsük el a verem adattípus specifikációját. Legyen az adattípus neve stack.

MEGOLDÁS

stack(data, n: **nat**) **is a class specification** =

parameters = bool + nat +

sorts: data

oprs: $\perp$: $\rightarrow$ data

exports =

type sorts: stack

oprs: create: $\rightarrow$ stack

push: stack data $\rightarrow$ stack

pop: stack $\rightarrow$ stack

top: stack $\rightarrow$ data

length: stack $\rightarrow$ nat

eqns: s$\in$stack; a, b$\in$data;

pop(create) = ceate

top(create) = $\perp$

length(create) = zerus

pop(push(s, a)) = s

top(push(s, a)) = a

length(push(s, a)) = succ(length(s))

length(push(s, a)) ≥ n ⇒ push(s, a) = "undefined"

end stack;

3.4. PÉLDA Adva van a sor következő informális specifikációja.

Konstrukciós műveletek:

init: üres sor előállítása.

add(q, e): egy e elem elhelyezése a q sor végén.

Nem konstrukciós műveletek:

rem(q): az első elemétől megfosztott sor előállítása q-ból.

head(q): a q sorban első helyen álló elem előállítása.

length(q): a q sorban lévő elemek számát adja meg eredményül.

A sorban lévő elemek száma nem lehet nagyobb, mint n, n > 0. Készítsük el a sor adattípus specifikációját. Legyen az adattípus neve que.

MEGOLDÁS

que (data, n: **nat**) **is a type specification** =

parameters = bool + nat +

sorts: data

oprs: ⊥: → data

exports =

type sorts: que

oprs: init: → que

add: que data → que

rem: que → que

head: que → data

length: que → nat

eqns: q∈que; a, b∈data;

rem(init) = init

head(init) = ⊥

length(init) = zerus

rem(add(q, a) = **if** q = init **then** q **else** rem(q) **fi**;

head(add(q, a)) = **if** q = init **then** a **else** head(q) **fi**;

length(add(q, a)) = succ(length(q))

length(add(q, a)) ≥ n ⇒ add(q, a) = "undefined"

end que;

3.5. PÉLDA Adjuk meg a táblázat absztrakt specifikációját a következő, szóban megadott specifikációja alapján. Legyen a típus neve table.

Konstrukciós műveletek:

null: az üres táblázatot hozza létre.

assign(t, p, e): a t táblázatban a p kulcshoz az e adatot rendeli hozzá.

Nem konstrukciós műveletek:

delete(t, p): a t táblázat p kulcsához az "üres"(⊥) értéket rendeli hozzá. Ez a művelet tehát azonos az assign(t, p, ⊥) művelettel.

read(t, p): a t táblázat p kulcsához tartozó értéket olvassa ki.

length(t): a táblázatban lévő nem üres elemek száma, amely nem lehet nagyobb, mint n, n > 0.

MEGOLDÁS

table(nat, data, n: **natbool**) **is a type specification** =

parameters = natbool +

sorts: data

oprs: ⊥: → data

exports =

type sorts: table

oprs: null: → table

assign: table nat data → table

delete: table nat → table

read: table nat → data

length: table → natbool

eqns: t∈table; p,q∈nat; e∈data;

delete(null, e) = null

read(null, p) = ⊥

length(null) = zerus

delete(assign(t, p, e), q) = **if** p = q **then** delete(t, q)
else assign(delete(t, q), p, e) **fi**

read(assign(t, p, e), q) = **if** p = q **then** e
else read(t, q)**fi**

length(assign(t, p, e)) = succ(length(delete(t, p)))

length(assign(t, p, e)) > 0 ⇒
assign(t, p, e) = "undefined"

end table;

3.6. PÉLDA Legyen adva a lista következő informális specifikációja:

elem: a rendezett elemek halmaza.

a ≤ b: az a és b elemek között fennálló rendezettség relációja.

a = b: két elem egyenlőségének a relációja.

list: az elemek egy véges listája.

Konstrukciós műveletek:

[]: üres lista;

add(e, *l*): az *l* lista elejére elhelyezünk egy e elemet.

Nem konstrukciós műveletek:

is-ordered(l): igaz értéket ad akkor, ha az l lista rendezett, különben pedig hamis értéket kapunk.

Készítsük el a rendezett lista specifikációját a fenti forma szerint.

MEGOLDÁS

list(elem) **is a type specification** =

parameters: **bool** +

sorts: elem

oprs: =: elem elem $\rightarrow$ bool [infix]

$\leq$: elem elem $\rightarrow$ bool [infix]

eqns: e, e_1, e_2, $e_3 \in$ elem;

$e_1 = e_2 \wedge e_2 = e_3 \Rightarrow e_1 = e_3$

$e_1 = e_2 \Rightarrow e_2 = e_1$

$e \leq e$ = "true"

$e_1 \leq e_2$ = "true" $\wedge$ $e_2 \leq e_3$ = "true" $\Rightarrow$ $e_1 \leq e_3$ = "true"

$e_1 \leq e_2$ = "true" $\wedge$ $e_2 \leq e_1$ = "true" $\Rightarrow$ $e_1 = e_2$

exports:

type sorts: list,

oprs:

[]: $\rightarrow l$

add: elem list $\rightarrow$ list

is-ordered: list $\rightarrow$ bool

eqns: $l \in$ list;

is-ordered([]) = "true";

is-ordered(add(e, [])) = "true"

$e_1 \leq e_2$ = "true" $\Rightarrow$

is-ordered(add(e_1, add(e_2, l))) =

is-ordered(add(e_2, l)

$e_1 \leq e_2$ = "false" $\Rightarrow$

is-ordered(add(e_1, add(e_2, l))) = "false"

end list;

4. AZ ADATTÍPUS OSZTÁLYÁNAK SPECIFIKÁCIÓJA

A specifikációmorfizmust a következőkben az adattípusok osztályának a specifikációjánál használjuk fel. Az osztály specifikációja több részből áll, amelyek a specifikációmorfizmus felhasználásával tömörebb formában megadhatók.
Az adattípusnak van absztrakt és konkrét megjelenési formája. Az osztály ennek a két megjelenési formának az egyesítése.

DEFINÍCIÓ A típusosztály egy formális definíciója. Az adattípus osztálya egy

(par, exp, imp, bod, e, i, eb, ib),

nyolcas, ahol

par: specifikáció, a paraméterek tulajdonságának leírása.

exp: kitüntetett szortú specifikáció, a típusobjektumok halmazának és a rajta értelmezett műveletek szintaxisának és szemantikájának a definíciója.

imp: specifikáció, a típusobjektumok ábrázolásához más osztályoktól átvett szolgáltatások tulajdonságainak leírása.

bod: kitüntetett szortú specifikáció, a típusosztály ábrázolásának, megvalósításának definíciója.

e: specifikációmorfizmus, amely a paraméterek specifikációját az "exp" specifikációra képezi le. A mi esetünkben ez a morfizmus tartalma-

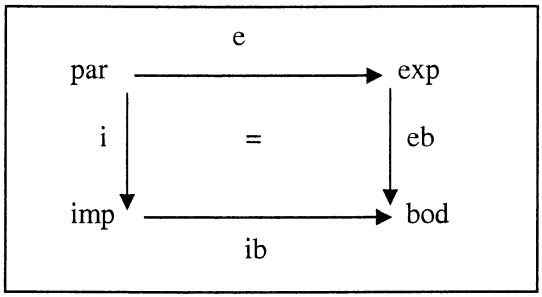
zást jelent, ami természetesen magába foglalja a bővítés lehetőségét is.

i: specifikációmorfizmus, amely a mi esetünkben szintén tartalmazást jelent; a paraméterek specifikációját az importfelület specifikációjára képezi le.

eb: kitüntetett szortú specifikációmorfizmus, amely az exportrészben létrejött specifikációt a törzsrész (body) specifikációjára képezi le. A mi esetünkben ez az absztrakt adattípus objektumok ábrázolásával együtt megvalósuló tartalmazást jelent.

ib: specifikációmorfizmus, amely az importfelület specifikációját a törzsrész specifikációjára képezi le. A mi esetünkben ez is tartalmazást jelent.

Az osztály morfizmusdiagramját a következő ábra szemlélteti:



ahol (eb o e) = (ib o i), ezt jelöli a morfizmusdiagramban az "=" jel.

A specifikációmorfizmus a mi esetünkben a következő lehet.

1. Üres.
2. Identitás.
3. Újraelnevezés:
   - A szort új nevet kap;

sort =

<új név> = <régi név>

- Az operáció új nevet kap;

oprs =

<új opr. név> = <régi opr.név>

4. Tartalmazás:
   - szortok esetében bővül a szortok listája;
   - operációk esetében újabb operációszimbólumokkal bővül a lista;
   - újabb axiómákkal bővül a lista, az axiómákban a változók helyén az új nevű ill. új szortok példányai is megjelenhetnek.
5. Ábrázolás:
   - A class szort helyére más típusok szortjaiból alkotott Descartes-szorzat lép;

oprs =

rep: <Descartes-szorzat> → <változó neve>

   - Az operációszimbólumokban a változók helyére a Descartes-szorzat változói lépnek;
   - Az axiómákban a szortok változóinak példányai helyére a Descartes-szorzat példányai kerülnek,

4.1. A TÍPUSOSZTÁLY FORMALIZÁLT LEÍRÁSA

⟨az osztály neve⟩ **is a class specification =**

parameters =<az átvett aktuális típusok nevei> +

sorts: <a formális paraméterek nevei>;

oprs: <a műveletek formái>;

eqns: <a műveletek jelentésének leírása>;

exports =

class sorts: <a típushalmaz neve>;

oprs: <a műveletek formái>;

eqns: <a műveletek jelentésének leírása>;

imports =

sorts: <az ábrázoláshoz importált típusok nevei>;

oprs: <a műveletek formái>;

eqns: <a műveletek jelentésének leírása>;

body =

oprs: rep: <az ábrázolás formája>;

eqns: <a jelentés konkrét leírása>;

end ⟨az osztály neve⟩;

4.1.1. PÉLDA Készítsük el a halmaz típusosztály formális specifikációját a következő szóbeli specifikáció alapján:

empty: az üres halmazt állítja elő.

insert(s, e): az s halmazban elhelyezi az e elemet. Az s halmaznak csak egyszeres elemei lehetnek.

delete(s, e): az s halmazból eltávolítja az e elemet.

has(s, e): arra adja meg a választ, hogy az s halmaz tartalmazza-e az e elemet.

size(s): az s halmazban lévő elemek számát adja meg.

A halmazban lévő elemek számának maximuma n, $n > 0$.

A halmaz ábrázolása egy vektorral és egy mutatóval történik:

$$\text{vector} \times \text{natbool} \rightarrow \text{set}.$$

MEGOLDÁS

set(data, n: **nat**) **is a class specification** =

parameters = bool + nat +

sorts: data

oprs: $\perp$: $\rightarrow$data

=: data data $\rightarrow$ bool [infix]

eqns: a, b, c$\in$ data;

a = b $\Rightarrow$ b = a

a = b $\wedge$ b = c $\Rightarrow$ a = c

exports =

class sorts: set

oprs: empty: $\rightarrow$ set

insert: set data $\rightarrow$ set

delete: set data $\rightarrow$ set

has: set data $\rightarrow$ bool

size: set $\rightarrow$ nat

eqns: s$\in$ set; a, b$\in$ data;

delete(empty, a) = empty

has(empty, a) = "false"

size(empty) = zerus

delete(insert(s, a), b) = **if** a = b **then** delete(s, b)
else insert(delete (s, b), a)**fi**

has(insert(s, a), b) = **if** a = b **then** "true"
else has(s, b) **fi**

size(insert(s, a)) = **if** has(s, a) **then** size(s)
else succ(size(s)) **fi**

size(insert(s, a)) > n
$\Rightarrow$ insert(s, a) = "undefined"

imports =

sorts: vector

oprs: nil: → vector

put: vector nat data → vector

access: vector nat → data

eqns: v∈ vector, i, j∈ nat, a∈ data;

access(nil, a) = ⊥

access(put(v, i, a), j) = **if** i = j **then** a

else access(v, j) **fi**

body =

oprs: rep: vector nat → set

eqns: v∈ vector, m∈ nat, a∈ data

empty= (nil, zerus)

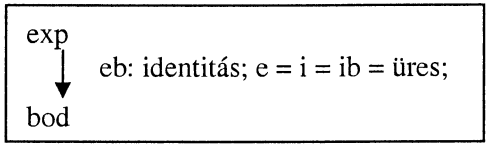
insert((v, m), a) = **if** (∃*l*, 1≤ *l* ≤ m)(access(v, *l*) = a

then (v, m) **else** (put(v, succ(m), a), succ(m))

end set;

4.2. AZ ADATTÍPUSOK OSZTÁLYÁNAK JELLEGZETES SPECIÁLIS ESETEI

1. Nincs paramétere és nincs importfelülete az adattípusnak, és a konkrét megjelenési forma kívülről látható.



4.2.1. PÉLDA

nat1 **is a class specification** =

parameters =

exports =

class sorts: nat1

oprs: zerus: → nat1

succ: nat1 → nat1

prec: nat1 → nat1

eqns: n∈nat1;

prec(zerus) = "undefined"

prec(succ(n)) = n

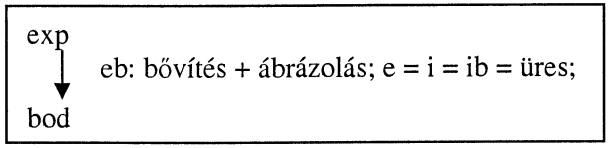
imports =

body =

oprs: rep: nat1 ≡ nat1

end nat1;

2. Nincs paramétere és nincs importfelülete az adattípusnak, és a konkrét megjelenési forma kívülről nem látható. A konkrét megvalósítás rejtve marad, beburkolás.



4.2.2. PÉLDA

nat2 **is a class specification** =

parameters =

exports =

class sorts: nat2

oprs: zerus: $\rightarrow$ nat2

succ: nat2 $\rightarrow$ nat2

prec: nat2 $\rightarrow$ nat2

eqns: n$\in$nat2;

prec(zerus) = "undefined"

prec(succ(n)) = n

imports =

body =

oprs: 0: $\rightarrow N$

+ 1: $\rightarrow N$

- 1: $\rightarrow N$

rep: nat $\rightarrow$ nat2;

eqns: k, m$\in N$

0 - 1 = "undefined"

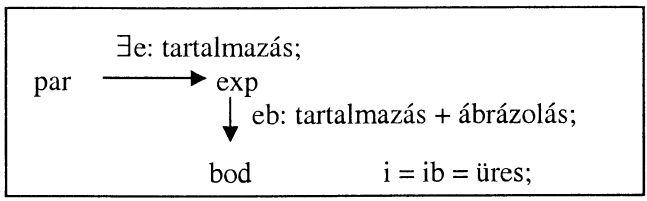
(k + 1) - 1 = k

zerus = 0

succ(m) = m + 1

end nat2;

3. Van paramétere, de nincs importfelülete az adattípusnak, és a konkrét megjelenési forma kívülről nem látható. A konkrét megvalósítás rejtve marad (beburkolás).



4.2.3. PÉLDA

verem(elem, n: **natbool**) **is a class specification** =

parameters = **natbool** +

sorts: elem

oprs: ⊥: → elem

exports:

class sorts: verem

oprs: create: → verem

push: verem elem → verem

pop: verem → verem

top: verem → elem

length: verem → natbool

eqns: v∈ verem; e∈elem;

pop(create) = create

top(create) = ⊥

length(create) = zerus

pop(push(v, e)) = v

top(push(v, e)) = e

length(push(v, e)) = succ(length(v))

length(v) ≥ n ⇒ push(v, e) = "undefined"

imports =

body =

sorts: vector

oprs: new: → vector

put: vector natbool elem → vector

access: vector natbool → elem

$length_c$: vector → natbool

rep: vector natbool → verem

eqns: v∈ vector; j, k, m∈ natbool; e∈ elem;

access(new, j) = ⊥

access(put(v, j, e), k) = **if** j = k **then** e

else access(v, k) **fi**;

$length_c$(new) = zerus

$length_c$(put(v, j, e)) = succ($length_c$(v))

create = (new, zerus)

push(v, m, e) = (put(v, succ(m), e), succ(m))

pop(v, m) = (v, prec(m))

top(v, m) = access(v, m)

length(v, m) = $length_c$(v)

end verem;

4.2.4. PÉLDA Készítsük el a lista osztály formális specifikációját a következő informális specifikáció alapján. Legyen a típusosztály neve list.

Az ábrázolás: vector natbool → list.

[ ] üres lista létrehozása.

[e] egy "e" elemből álló lista létrehozása.

add(e, n) egy "e" elem elhelyezése az "n" lista elején.

n . m az "n" lista után az "m" lista elhelyezése.

head(n) a lista első elemének a kiolvasása.

rest(n) az első elemétől megfosztott lista maradékának a kiolvasása.

e∈n arra a kérdésre válaszol, hogy "e" szerepel-e az "n" listában.

isprefix(n, m) arra a kérdésre válaszol, hogy az "n" lista kezdete-e az "m" listának.

ispostfix(n, m) arra a kérdésre válaszol, hogy az "n" lista végét képezi-e az "m" listának.

pred(e, e', n) arra a kérdésre válaszol, hogy "e" megelőzi-e sorrendben az "e'" elemet az "n" listában.

eg(n, m) arra a kérdésre válaszol, hogy az n és az m lista egyenlők-e. Az egyenlőség azt jelenti, hogy az elemeik rendre megegyeznek.

reverse(n) az n listában lévő elemeket fordított sorrendben helyezi el az eredménylistában.

| _ |: a listában lévő elemek száma, amely legfeljebb N pozitív egész szám lehet.

MEGOLDÁS

list(elem, N: **natbool**) **is a class specification** =

parameters = natbool +

sorts: elem

oprs: ⊥: elem

=: elem elem → bool

eqns: e, e', e"∈elem;

e = e' ⇒ e' = e

e = e' ∧ e' = e" ⇒ e= e"

exports =

class sorts: lista

oprs: [ ]: → list

[_]: elem → list

add: elem list → list

_ . _: list list→ list

head: lista → elem

rest: list → list

∈: elem list → bool [infix]

isprefix: list list → bool

pred: elem elem list → bool

reverse: list → list

ispostfix:list list → bool

| _ |: list → natbool

eqns: e, e', e"∈elem; k, n, m, s∈lista;

[e] = add(e, [ ])

[ ] . n = n

add(e, n) . m = add(e, n . m)

head([ ]) = ⊥

head(add(e, n))= e

rest([ ]) = [ ]

rest(add(e, n)) = n

e∈[ ] = "false"

e∈add(e', n) = **if** e = e' **then** "true" **else** e∈n **fi**

isprefix([ ], n) = "true"

isprefix(add(e, n), [ ]) = "false"

isprefix(add(e, n), add(e', m)) =

if e = e' ∧ isprefix(n, m) **then** "true" **else** "false" **fi**

pred(e, e', [ ]) = "false"

pred(e, e', add(e", n)) =

if ((e = e" ∧ e'∈n) ∨ pred(e, e', n))

then "true" **else** "false" **fi**

eq(n, m) ⇒ eq(m, n)

eq([ ], [ ]) = "true"

eq([ ], add(e, n)) = "false"

eq(add(e, n), add(e', m)) = (e = e' ∧ eq(n, m))

reverse([ ]) = [ ]

reverse(add(e, n)) = reverse(n) . [e]

ispostfix(n, m) =

isprefix(reverse(n), reverse(m))

| [ ] | = zerus

|add(e, n)| = succ(| n |)

|add(e, n)| > N ⇒ add(e, n) = "undefined"

imports =

sorts: vector

oprs: nil: →vector

put: vector natbool elem → vector

access: vector natbool → elem

$shift_r$: vector → vector

eqns: v∈ vector; i, j∈ natbool; e∈ elem;

access(nil, j) = ⊥

access(put(v, i, e), j) = **if** i = j

then e **else** access(v, j) **fi**

$shift_r$(nil) = nil

$shift_r$(put(v, j, e)) = put($shift_r$(v), succ(j), e)

body =

oprs: rep: vector natbool → list

eqns: v∈ vector; k∈ natbool; e∈ elem;

[ ] = (nil, zerus)

add(e, (v, k)) = **if** k < N

then (put($shift_r$(v), succ(zerus), e), succ(k))

else "undefined" **fi**

end list;

4.2.5. PÉLDA Készítsük el a kupac típusosztályának specifikációját a következő informális specifikáció alapján. Legyen a kupac típus neve heap.

hcreate: az üres kupac létrehozása;

hpush(h, e): a h kupacba az e elem behelyezése;

hpop(h): a h kupacból a legnagyobb elem kitörlése;

htop(h): h kupacból a legnagyobb elem kiolvasása;

length(h): a h kupacban lévő elemek száma;

A kupacban lévő elemek számának maximuma n; n > 0;

Ábrázoljuk a kupacot egy vektorral és egy mutatóval:

vector × natbool → heap;

MEGOLDÁS

heap(data, n: **nat**) **is a class specification** =

parameters = bool + nat +

sorts: data

oprs: $\perp$: $\rightarrow$ data

=: data data $\rightarrow$ bool [infix]

$\leq$: data data $\rightarrow$ bool [infix]

>: data data $\rightarrow$ bool [infix]

eqns: a, b, c$\in$ data;

$a = b \Rightarrow b = a$

$a = b \wedge b = c \Rightarrow a = c$

$a > b \wedge b > c \Rightarrow a > c$

$a > b$ = "true" $\Rightarrow a = b$ = "false"

$a \geq b \wedge b \geq c \Rightarrow a \geq b$

$a \geq b \wedge b \geq a \Rightarrow a = b$

exports =

class sorts: heap

oprs: hcreate: $\rightarrow$ heap

hpush: heap data $\rightarrow$ heap

hpop: heap $\rightarrow$ heap

htop: heap $\rightarrow$ data

length:heap $\rightarrow$ nat

eqns: h$\in$ heap; a$\in$ data;

hpop(hcreate) = hcreate

htop(hcreate) = $\perp$

length(hcreate) = zerus

htop(hpush(h, a) = a

$htop(h) \leq a \Rightarrow$ htop(hpush(h, a) = a

$htop(h) > a \Rightarrow$ htop(hpush(h, a) = htop(h)

hpop(hpush(hcreate, a)) = hcreate

htop(h) ≤ a ⇒ hpop(hpush(h, a) = h

htop(h) > a ⇒

hpop(hpush(h, a) = hpush(hpop(h), a)

length(hpush(h, a)) = succ(length(h))

length(hpush(h, a)) ≥ n ⇒

hpush(s, a) = "undefined"

imports =

sorts: vector

oprs: new: → vector

put: vector nat data → vector

access: vector nat → data

$shift_R$: vector nat → vector

eqns: v∈ vector; i, j∈ nat, a∈ data;

access(new, i) = ⊥

$shift_R$(new, i) = new

i = j ⇒ access(put(v, i, a), j) = a

i ≠ j ⇒ access(put(v, i, a), j) = access(v, j)

i < j ⇒ $shift_R$(put(v, i, a), j) = put($shift_R$(v, j), i, a)

j ≤ i ⇒ $shift_R$(put(v, i, a), j) =

put($shift_R$(v, j), succ(i), a)

body =

oprs: rep: vector nat →heap

eqns: v∈ vector; m, j∈ nat, a∈ data; 1= succ(zerus);

hcreate = (new, zerus)

a ≤ access(v, 1) ⇒

hpush(v, m, e) = (put($shift_R$(v, 1), 1, a), succ(m))

$(\exists j,\ 1< j \leq m)(access(v,\ prec(j)) \leq a < access(v,\ j))$

$\Rightarrow hpush(v, m, e) = (put(shift_R(v, j), j, a), succ(m))$

$access(v, m) < a \Rightarrow$

$hpush(v, m, e) = (put(v, succ(m), a), succ(m))$

end heap;

Az adattípusok ábrázolására gyakran több összetett adattípust is felhasználunk. Erre szolgál példaként a zsák.

4.2.6. PÉLDA Adott a zsák informális specifikációja:

bempty: üres zsák előállítása;

binsert(b, e): a b zsákban egy e elem elhelyezése, a zsákban egy elem többszörösen is előfordulhat.

bdelete(b, e): a b zsákból egy e elem kitörlése.

many(b, e): eredményül a b zsákban az e elem multiplicitását szolgáltatja.

bsize: a zsákban lévő összes elem száma.

Válasszunk két vektort és egy mutatót a zsák ábrázolására. Az egyik vektorban hézagmentesen egymás után elhelyezzük az elemeket, a másikban pedig azonos indexérték mellett az elemek multiplicitásának értékeit. A mutató egy nem negatív egész szám, amely azt mutatja, hogy hány különböző elem van a zsákban. Az ábrázolás tehát:

$$vector \times vectot \times natbool \rightarrow bag.$$

Készítsük el a zsák típusosztály specifikációját. A zsák típus neve legyen bag.

MEGOLDÁS

bag(data, n: **nat**) **is a class specification** =

parameters = nat +

sorts: data

oprs: ⊥: →data

=: data data → bool [infix]

eqns: a, b, c∈data;

a = b ⇒ b = a

a = b ∧ b = c ⇒ a = c

exports =

class sorts: bag

oprs: bempty: → bag

binsert: bag data → bag

bdelete: bag data → bag

many: bag data → nat

bsize: bag → nat

eqns: b∈bag; a, e∈data;

bdelete(bempty, a) = bempty

many(bempty, a) = zerus

bsize(bempty) = zerus

bdelete(binsert(b, a), e) = **if** a = e **then** b

else binsert(bdelete(b, e), a) **fi**

many(binsert(b, a), e) = **if** a = e

then succ(many(b)) **else** many(b, e) **fi**

bsize(binsert(b, a)) = succ(bsize(b))

bsize(binsert(b, a)) > n

⇒ binsert(s, a) = "undefined"

imports =

sorts: vector

oprs: nil:→ vector

put: vector nat data → vector

access: vector nat → data

$shift_L$: vector nat → vector

eqns: v∈ vector; i, j∈ nat; d∈ data;

access(nil, i) =⊥;

$shift_L$(nil) = nil;

access(put(v, i, d), j) =

if i = j **then** d **else** access(j) **fi**

i = j ⇒ $shift_L$(put(v, i, d), j) = $shift_L$(v, j);

1 < i < j ⇒

$shift_L$(put(v, i, d), j) = put($shift_L$(v, j), i, d);

i > j ⇒

$shift_L$(put(v, i, d), j) = put($shift_L$(v, j), prec(i), d);

body =

oprs: rep: vector vector nat → bag

eqns: v∈ vector(nat, data), u∈ vector(nat, nat),

m∈ nat, d∈ data; 0 = zerus; 1 = succ(zerus);

bempty = (nil, nil, 0);

binsert(v ,u, m, d) =

if (∃j, 1 ≤ j ≤ m)(access(v, j) = d)

then (v, put(u, j, succ(access(u, j)), succ(m)) **else**

(put(v, succ(m), d),put(u, succ(m), 1), succ(m)) **fi**

end bag;

A gyakorlatban általában összetett feladatokkal dolgozunk. Az ilyen feladatokban szereplő adattípusok azonban egyszerű adattípusokból tevődnek össze. Ilyenek például a többdimenziós adattípusok. Ennek a legegyszerűbb formája a

kétdimenziós tömb. A kétdimenziós tömbnek a sorai azonos hosszúságúak, mi azonban a következőkben olyan kétdimenziós szerkezeteket definiálunk, amelyekben az összetett adattípus objektumainak elemei maguk is összetett adattípus objektumok. Ezeket hiperszerkezeteknek nevezzük. A következőkben egy ilyen hiperszerkezet osztályának specifikációjára mutatunk példát.

4.2.7. PÉLDA Adjuk meg a veremtáblázat típusosztályának algebrai specifikációját a következő, szóban megadott specifikációja alapján. A veremtáblázat olyan táblázat, amelynek kulcsaihoz több érték is tartozhat, de egy adott kulcs esetén mindig csak az utolsónak lehelyezett értékhez férhetünk hozzá. Legyen a veremtáblázat neve stacktable.

stnull: az üres stacktáblázatot hozza létre.

stassign(st, p, e): az st stacktáblázatban a p kulcshoz az e adatot rendeli hozzá.

stdelete(st, p): az st stacktáblázat p kulcsához tartozó értékek közül az utolsónak lehelyezett értéket kitörli.

stread(st, p): az st stacktáblázat p kulcsához tartozó értékek közül az utolsónak lehelyezett értéket olvassa ki.

stlength(st): az st stacktáblázatban lévő összes elem száma.

Ábrázoljuk a veremtáblázatot hipervektorral, amelynek elemei vektorok. Legyenek a kulcsok nem negatív egész számok. Ezek alkossák a hipervektor indexét. A hipervektorra a szokásos formában definiáljuk a vektorműveleteket:

hnil;

hv[j]::= e;

hv[j];

A vektor esetében pedig használjuk a műveletek prefix formáját:

nil;

put(v, i, e);

access(v, j);

MEGOLDÁS

stacktable(nat, data, n: **natbool**) **is a class specification** =

parameters = natbool +

sorts: data

oprs: $\perp$: $\rightarrow$ data

exports =

class sorts: stacktable

oprs: stnull: $\rightarrow$ stacktable

stassign: stacktable nat data $\rightarrow$ table

stdelete: stacktable nat $\rightarrow$ table

stread: stacktable nat $\rightarrow$ data

stlength: stacktable $\rightarrow$ nat

eqns: st$\in$ stacktable; p, q$\in$ nat; e$\in$ data;

stdelete(stnull, e) = stnull

stread(stnull, p) = $\perp$

stlength(stnull) = zerus

stdelete(stassign(st, p, e), q) = **if** p = q **then** st

else stassign(stdelete(st, q), p, e) **fi**

stread(stassign(st, p, e), q) = **if** p = q **then** e

else stread(st, q) **fi**

stlength(stassign(st, p, e)) =

succ(stlength(stdelete(st, p)))

stlength(st) $\geq$ n $\Rightarrow$

stassign(st, p, e) = "undefined"

imports =

sorts: hvector, vector

oprs: nil: → vector

put: vector natbool data → vector

access: vector natbool → data

length: vector → natbool

hnil: → hvector

_ [_]::= _: hvector natbool vector → hvector

_ [_]: hvector natbool → vector

eqns: v vector; hv∈hvector; e∈data; j, k∈natbool;

access(nil, j) = ⊥

access(put(v, j, e), k) = **if** j = k

then e **else** access(v, k) **fi**;

length(nil) = 0

length(put(v, j, e)) = **if** access(v, j) = ⊥

then length(v) + 1 **else** length(v) **fi**;

[hnil, j] = nil

(hv[j] ::= v)[k] = **if** j = k **then** v **else** hv[k] **fi**;

body =

oprs: φ: hvector natbool → stacktable

eqns: hv∈hvector; m, j∈natbool; e∈data;

m = 0 ⇒ φ(hv, m) = stnull

(∃j, hv[j] ≠ nil) ⇒ φ(hv, m) =

stassign((φ(hv[j] ::= put(hv[j], length(hv[j]), ⊥),

m - 1)), j, access(hv[j], length(hv[j])))).

end stacktable;

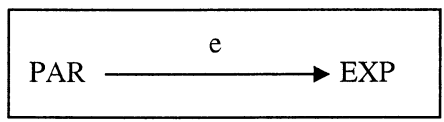
5. A TÍPUSOSZTÁLY SPECIFIKÁCIÓJÁNAK ELEMZÉSE

Az osztály specifikációjának elemzése során a következő kérdésekre keressük a választ:

- Jó-e az absztrakt leírás? Az osztály szolgáltatásai megfelelnek-e a felhasználó elvárásainak? Erre a kérdésre az absztrakt specifikáció elemzése kívánja megadni a választ.
- Jó-e az ábrázolás? Helyesen ábrázoltuk-e az absztrakt típusobjektumot? Ennek a vizsgálatát célozza a reprezentáció elemzése.
- Jó-e a megvalósítás? Helyesen implementáltuk-e az adattípus szolgáltatásait jelentő műveleteket? Ez a tárgya az implementáció elemzésének.

5.1. AZ ABSZTRAKT SPECIFIKÁCIÓ ELEMZÉSE

Legyen adott az exportfelület absztrakt specifikációja:



Formálisan:

$$\Sigma = (S, OP); \Sigma = (S_A, OP_A); SPEC_a = (S_A, OP_A, E_A),$$

ahol a szemantika algebrai axiómákkal van megadva.

Mi a következőkben a kényelmesebb jelölést használjuk:

$$SPEC_a = d_a = (A, F, E_a),$$

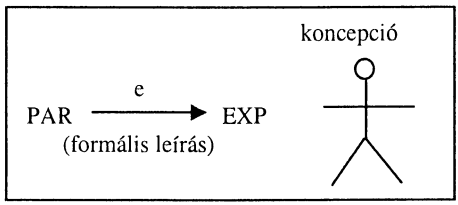
részletesen:

$$d_a = (\{A_0, A_1, \ldots, A_n\}, \{f_0: \rightarrow A_0, \ldots, f_m: A_i \ldots A_j \rightarrow A_k\},$$

$$\{\ldots, \alpha(a) \Rightarrow f_s(f_c(a)) = h(a), \ldots\}),$$

ahol $a \in A$, azaz tetszőleges $(a_i, \ldots, a_k) \in A_i \times \ldots \times A_k$.

Kérdés, hogy az absztrakt specifikáció az általunk elképzelt koncepció szerint jó-e. Adva van tehát egy koncepció az adattípus szolgáltatásairól, és adva van ezeknek a szolgáltatásoknak a szintaktikailag és szemantikailag egyértelmű leírása. Ezt szemlélteti a következő ábra.



A feltett kérdésre a választ a specifikáció elemzésével kívánjuk megadni. Ez csak úgy lehetséges, ha a koncepció alapján meg tudjuk fogalmazni formálisan a szolgáltatások tulajdonságait, és bebizonyítjuk, hogy ezek a tulajdonságok levezethetők a mi formális rendszerünkből. Ez nehéz feladat. De a legfontosabb tulajdonságok felírása általában automatikusan történhet. A mi algebrai axiómáink általában olyanok, hogy a konstrukciós műveletek után alkalmazott szelekciós műveletek eredményeit rögzítik. Kézenfekvő tehát a koncepció alapján megfogalmazni két művelet egymás utáni alkalmazásának az eredményét azokra az esetekre, amelyek nem szerepelnek az axiómák között. Ezek a tételek tehát általában a következők lesznek:

$$f_c(f_c'(a)) = \ldots$$

$$f_s(f_s'(a)) = \ldots$$

Legyen

$$H_1(a) = H_2(a),$$

az absztrakt adattípus bizonyítandó tulajdonságát leíró tétel, ahol $H_1(a)$, $H_2(a)$ az $f_i \in F$, műveletek egymás utáni alkalmazásával felírt kifejezés. Általában két művelet egymás utáni alkalmazásáról van szó a $H_1(a)$ és $H_2(a)$ kifejezésben is.

- Ha a kifejezések értéke: $H_1(a)$, $H_2(a) \in A_0$, akkor a vizsgált típus két objektumának az egyenlőségét kell eldönteni:

  $$a_1 = a_2?$$

  Ez a vizsgált típusobjektumok terében eldöntendő kérdés.

- Ha a kifejezések értéke: $H_1(a)$, $H_2(a) \in A_i$, $i \geq 1$, akkor az adattípus valamelyik paraméterének két objektumáról kívánjuk bebizonyítani, hogy azok egyenlők:

  $$p_1 = p_2?$$

  Ez viszont a paraméterek terében eldöntendő kérdés.

Mit jelent az, hogy két absztrakt objektum egymással egyenlő? Ez definíció kérdése. Adjuk meg tehát ezt axiómaként.

DEFINÍCIÓ Az egyenlőség axióma. Az egyenlőségre általános formában a következő definíciót adjuk:

$$a_1 = a_2 \equiv [(a_1 = f_0) \wedge a_2 = f_0)] \vee [(\forall f_s \in F_s)(f_s(a_1) = f_s(a_2)]$$

Itt a vagy kizárólagos vagy-ot jelent.

Ez az egyenlőségnek egy rekurzív definíciója. Két összetett objektum egyenlőségét visszavezettük azok részeinek az egyenlőségére. Két megkonstruált objektumot tehát akkor tekintünk egyenlőnek, ha azok ugyanazokkal a szelekciós műveletekkel bonthatók le az üres objektumra úgy, hogy közben a kapott ismert típusú részeredmények is egyenlők.

5.1.1. PÉLDA A halmaz egyenlőség axiómája:

$$s_1 = s_2 \equiv (s_1 = \text{empty} \wedge s_2 = \text{empty}) \vee$$
$$[(\forall e \in \text{elem})(\text{has}(s_1, e) = \text{has}(s_2, e) \wedge \text{delete}(s_1, e) = \text{delete}(s_2, e)) \wedge$$
$$\text{size}(s_1) = \text{size}(s_2)]$$

Az egyenlőség axióma egy általános formulát fogalmaz meg. Ez az általános formula rendszerint egyszerűsíthető. Például a halmaz esetében ilyen egyszerűsített formula lehet a következő:

$$s_1 = s_2 \equiv (s_1 = \text{empty} \wedge s_2 = \text{empty}) \vee (\forall e \in \text{elem})(\text{has}(s_1, e) = \text{has}(s_2, e)).$$

Kézenfekvő azt gondolni, hogy ha ezzel a formulával definiáljuk az egyenlőséget, akkor

$$s_1 = s_2 \Rightarrow (s_1 = \text{empty} \wedge s_2 = \text{empty}) \vee$$
$$(\forall e \in \text{elem})(\text{delete}(s_1, e) = \text{delete}(s_2, e) \wedge \text{size}(s_1) = \text{size}(s_2))$$

Azt a kérdést azonban joggal fel kell tenni, hogy ez a mi formális rendszerünkből is következik-e. A specifikáció elemzésének tehát ez is részét képezi.

Az egyenlőség axióma alkalmazásával, az adattípus szemantikáját meghatározó axiómák felhasználásával és – általában – strukturális indukcióval bizonyítható be két típusobjektum

$$f_i(\ldots f_j(a)\ldots) = f_k(\ldots f_l(a)\ldots)$$

formulával adott egyenlősége.

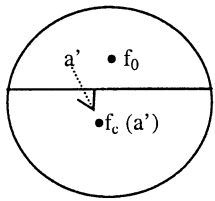
Tekintsük az egyenlőség bal oldalát.

1. Ha $f_i = f_c \in F_c$ konstrukciós művelet, akkor alkalmazva az egyenlőség axiómát

   $$f_s(f_c(\ldots)$$

   szemantikai axióma lesz.

2. Ha viszont $f_i \notin F_c$, viszont $f_j \in F_s$ szelekciós művelet, akkor alkalmazhatjuk a strukturális indukciót, amit a következő ábra szemléltet:



- alapeset: $a = f_0$, és ekkor $f_s(f_0)$ szemantikai axióma és erre az esetre bizonyítjuk a tételt.
- indukciós lépés: $a = f_c(a')$, ahol a'-re már igaz a tétel állítása. Ekkor $f_s(f_c(a'))$ szintén szemantikai axióma, és erre az esetre is bizonyítjuk a tételt.

5.1.2. PÉLDA Bizonyítsuk be a "nat" típus estén, hogy

$$m \neq \text{zerus} \Rightarrow \text{add}(n, \text{pre}_c(m)) = \text{pre}_c(\text{add}(n, m)).$$

BIZONYÍTÁS Az egyenlőség axiómát felírva:

$$n_1 = n_2 \equiv (n_1 = \text{zerus} \wedge n_2 = \text{zerus}) \vee (\text{pre}_c(n_1) = \text{pre}_c(n_2)).$$

Legyen $m = \text{succ}(m')$; $m' \geq \text{zerus}$, amelyre már az állítás igaz.

a bal oldal:

$$\text{add}(n, \text{pre}_c(\text{succ}(m'))) = \text{add}(n, m').$$

a jobb oldal:

$$\text{pre}_c(\text{add}(n, \text{succ}(m'))) = \text{pre}_c(\text{succ}(\text{add}(n, m'))) = \text{add}(n, m').$$

5.1.3. PÉLDA Bizonyítsuk be a halmaz esetében, hogy

$$\text{insert}(\text{insert}(s, e), e) = \text{insert}(s, e).$$

BIZONYÍTÁS Felhasználva két halmaz egyenlőségének egyszerűsített axiómáját, azt kell bizonyítani, hogy

$$\text{has}(\text{insert}(\text{insert}(s, e), e), a) = \text{has}(\text{insert}(s, e), a).$$

a bal oldal:

has(insert(insert(s, e), e), a) =

if e = a **then** "true" **else** has(insert(s, e), a) **fi** =

if e = a **then** "true" **else** has(s, a) **fi;**

a jobb oldal:

has(insert(s, e), a) =

if e = a **then** "true" **else** has(s, a) **fi;**

5.1.4. PÉLDA Bizonyítsuk be a halmaz típus esetében a következő tételt:

$$\sim has(h, e) \Rightarrow delete(h, e) = h$$

BIZONYÍTÁS Legyen h = empty; Ekkor

a bal oldal:

~has(empty, e) = "true";

a jobb oldal:

delete(empty, e) = empty,

tehát a tétel állítása ebben az esetben igaz.

Legyen h = insert(h', e'), ahol feltesszük, hogy

$$\sim has(h', e) \Rightarrow delete(h', e) = h'.$$

Ekkor

a bal oldal:

$e = e' \Rightarrow \sim has(insert(h', e'), e) =$ "false";

a jobb oldal:

$e = e' \Rightarrow delete(insert(h', e'), e) = delete(h', e);$

a bal oldal:

$e \neq e' \Rightarrow \sim has(insert(h', e'), e) = \sim has(h', e);$

a jobb oldal:

$e \neq e' \Rightarrow$

delete(insert(h', e'), e) = insert(delete(h', e), e') = insert(h', e'),

tehát az állítás igaz.

5.1.5. PÉLDA Bizonyítsuk be a halmaz típus esetében, hogy

delete(delete(s, e), e) = delete(s, e).

BIZONYÍTÁS Stukturális indukció és az egyenlőség axióma felhasználásával bizonyíthatjuk a tételt.

Alapeset: s = empty

a bal oldal:

delete(delete(empty, e), e) = delete(empty, e) = empty;

a jobb oldal:

delete(empty, e) = empty.

Indukció: s = insert(s', e'), ahol s'-re igaz az állítás.

a bal oldal:

$e = e' \Rightarrow$ delete(delete(insert(s', e'), e), e) =

delete(delete(s', e'), e) = delete(s', e');

$e \neq e' \Rightarrow$ delete(delete(insert(s', e'), e), e) =

delete(insert(delete(s', e), e'), e) =

insert(delete(delete(s', e), e), e') = insert(delete(s', e), e');

a jobb oldal:

$e = e' \Rightarrow$ delete(insert(s', e'), e) = delete(s', e');

$e \neq e' \Rightarrow$ delete(insert(s', e'), e) = insert(delete(s', e), e').

Megjegyzés az egyenlőség axióma egyszerűsítéséhez.

ÁLLÍTÁS Ha egyenlőség axiómának az

$s_1 = s_2 \equiv (s_1 = \text{empty} \wedge s_2 = \text{empty}) \vee (\forall e \in \text{elem})(\text{has}(s_1, e) = \text{has}(s_2, e))$

axiómát fogadjuk el, akkor

$$s_1 = s_2 \Rightarrow (\forall e \in elem)(delete(s_1, e) = delete(s_2, e)).$$

BIZONYÍTÁS Tegyük fel, hogy

$$has(delete(s, a), b) = \textbf{if } a = b \textbf{ then } \text{"false"} \textbf{ else } has(s, b) \textbf{ fi;}$$

Ekkor

$$has(delete(s_1, e), a) = \textbf{if } e = a \textbf{ then } \text{"false"} \textbf{ else } has(s_1, a) \textbf{ fi;}$$

és

$$has(delete(s_2, e), a) = \textbf{if } e = a \textbf{ then } \text{"false"} \textbf{ else } has(s_2, a) \textbf{ fi;}$$

azaz igaz az állítás.

MOST BIZONYÍTSUK A FELTEVÉST Bizonyítandó, hogy

$$a = b \Rightarrow has(delete(s, a), b) = \text{"false"}$$

$$a \neq b \Rightarrow has(delete(s, a), b) = has(s, b).$$

A bizonyítás strukturális indukcióval történhet.

Alapeset: s= empty.

$$a = b \Rightarrow has(delete(empty, a), b) = has(empty, b) = \text{"false"}.$$

$$a \neq b \Rightarrow has(delete(empty, a), b) = has(empty, b) = \text{"false"}.$$

Indukciós lépés: Ha s' = insert(s, c) és has(s, c)="false", akkor s-re igaz a tétel állítása.

$$a = b \Rightarrow has(delete(insert(s, c), a), b) =$$

$$\textbf{if } c = a \textbf{ then } has(delete(s, a), b) \textbf{ else } has(insert(delete(s, a), c), b) \textbf{ fi} =$$

$$\textbf{if } c = a \textbf{ then } has(s, b) \textbf{ else } has(insert(s, c), b) \textbf{ fi;}$$

Az eredmény tehát:

$$a = b = c \Rightarrow has(delete(s', a), b) = \text{"false"};$$

$$a = b \neq c \Rightarrow has(delete(s', a), b) = has(s', b)$$

$$a \neq b \Rightarrow has(delete((insert(s, c), a), b) = \textbf{if } c = a \textbf{ then } has(delete(s, a), b)$$

$$\textbf{else } has(insert(delete(s, a), c), b) \textbf{ fi;}$$

azaz

$$a \neq b \wedge c = a \Rightarrow has(delete(s', a), b) = has(delete(s, a), b) =$$

has(s, b) = has(s', b),

vegyük észre, hogy most b ≠ c

a ≠ b ∧ c ≠ a ⇒ has(insert(delete((s, a), c), b) = **if** c = b **then** "true"

else has(delete(s, a), b) **fi;**

ámde

has(delete(s, a), b) = has(s, b) = has(s', b).

5.1.6. PÉLDA Bizonyítsuk be, hogy a halmaz típus esetében

size(delete(s, e)) = **if** has(s, e) **then** prec(size(s)) **else** size(s) **fi;**

BIZONYÍTÁS Axióma:

size(insert(s, e)) = **if** has(s, e) **then** size(s) **else** succ(size(s)) **fi;**

másképpen

size(insert(s, e)) = succ(size(delete(s, e)))

ezért

prec(size(insert(s, e))) = size(delete(s, e))

Behelyettesítve size(insert(s, e)) formuláját, előáll a megoldás.

MÁSIK MEGOLDÁS Az eredeti axióma alapján strukturális indukcióval kissé bonyolultabb a bizonyítás.

Alapeset: s = empty

a bal oldal:

size(delete(empty, e)) = size(empty) = zerus;

a jobb oldal:

has(empty, e) = "false", ezért size(empty) = zerus.

Indukció: s' = insert(s, a), ahol has(s, a) = "false".

size(s') = size(insert(s, a)) = **if** has(s, a) **then** size(s) **else** succ(size(s)) **fi;**

has(s', e) ⇒ size(delete(s', e)) = size(delete(insert(s, a), e)) =

if has(delete(s, e), a) **then** size(delete(s, e)) **else** succ(size(delete(s, e))) **fi;**

tehát

$$e = a \Rightarrow size(delete(s', e)) = prec(size(s'));$$

$$e \neq a \Rightarrow size(delete(s', e)) = succ(size(delete(s, e))) =$$

if has(s, e) **then** succ(prec(size(s))) **else** succ(size(s)) **fi;** =

if has(s', e) **then** prec(size(s')) **else** size(s') **fi**;

5.1.7. PÉLDA Egészítsük ki a halmaz esetében a műveletek listáját a

$$sub: set\ set \Rightarrow set$$

művelettel, amelynek jelentése az, hogy az első halmazból kivonjuk azokat az elemeket, amelyek a második halmazban is benne vannak.

1. Adjuk meg először a sub jelentését meghatározó axiómákat:

$$sub(s, empty) = s$$

$$sub(empty, s') = empty$$

$$sub(s, insert(s', e)) = delete(sub(s, s'), e)$$

$$sub(insert(s, e), s') = \textbf{if}\ has(s', e)\ \textbf{then}\ sub(s, s')\ \textbf{else}\ insert(sub(s, s'), e)\ \textbf{fi;}$$

2. Bizonyítsuk be a következő tételt:

$$has(sub(s, s'), e) = has(s, e) \wedge \sim has(s', e).$$

BIZONYÍTÁS A bizonyítás strukturális indukcióval történhet s' szerint tetszőleges s mellett.

Ehhez felhasználjuk a

$$has(insert(s, a), b) = \textbf{if}\ a = b\ \textbf{then}\ \text{"true"}\ \textbf{else}\ has(s, b)\ \textbf{fi;}$$

axiómát, a már bizonyított

$$has(delete(s, a), b) = \textbf{if}\ a = b\ \textbf{then}\ \text{"false"}\textbf{else}\ has(s, b)\ \textbf{fi;}$$

tételt és a

$$b_1 = b_2 \equiv (b_1 = \text{"true"} \wedge b_2 = \text{"true"}) \vee (b_1 = \text{"false"} \wedge b_2 = \text{"false"})$$

egyenlőség axiómát.

Alapeset: s' = empty,

a bal oldal:

has(sub(s, empty), e) = has(s, e);

a jobb oldal:

has(s, e) ∧ ~has(empty, e) = has(s, e).

Indukció: s' = insert(s_1, a)

a bal oldal:

has(sub(s, insert(s_1, a)), b) = has(delete(sub(s, s_1), a), b) =

if a = b **then** "false"**else** has(sub(s, s_1), b) **fi**;

a jobb oldal:

has(s, b) ∧ ~has(insert(s_1, a), b) = **if** a = b

then has(s, b) ∧ ~"true"**else** has(s, b) ∧ ~has(s_1, b) **fi** =

if a = b **then** "false" **else** has(sub(s, s_1), b) **fi**;

5.1.8. PÉLDA Legyen adva a táblázat típusspecifikációja. Végezzük el a specifikáció elemzését! Fogalmazzuk meg a tételeket és bizonyítsuk be azokat!

table(nat, data) **is a type specification** =

parameters = natbool +

sorts: data

oprs: ⊥: ⟶ data

exports =

class sorts: table

oprs: null: ⟶ table

assign: table nat data ⟶ table

delete: table nat ⟶ table

read: table nat ⟶ data

length: table ⟶ nat

eqns: t∈table; p, q∈nat; e∈data;

delete(null, e) = null

read(null, p) = ⊥

length(null) = zerus

delete(assign(t, p, e), q) = **if** p = q **then**

delete(t, q) **else** assign(delete(t, q), p, e) **fi**

read(assign(t, p, e), q) =

if p = q **then** e **else** read(t, q) **fi**

length(assign(t, p, e)) = succ(length(delete(t, p)))

end table;

AZ ABSZTRAKT SPECIFIKÁCIÓ ELEMZÉSE

Írjuk fel az egyenlőség axiómát:

$t_1 = t_2 \equiv (t_1 = \text{null} \wedge t_2 = \text{null}) \vee (\forall p, 1 \leq p \leq n)(\text{read}(t_1, p) = \text{read}(t_2, p))$.

ÁLLÍTÁS $p \neq q \Rightarrow$ assign(assign(t, p, a), q, e) = assign(assign(t, q, e), p, a).

BIZONYÍTÁS Alkalmazzuk mindkét oldalra az egyenlőség axiómát.

a bal oldal:

read(assign(assign(t, p, a), q, e), j) =

if j = q **then** e **else** read(assign(t, p, a), j) **fi**;

read(assign(t, p, a), j) = **if** j=p **then** a **else** read(t, j) **fi**;

a jobb oldal:

read(assign(assign(t, q, e), p, a), j) =

if j = p **then** a **else** read(assign(t, q, e), j) **fi**;

read(assign(t, q, e), j) = **if** j = q **then** e **else** read(t, j) **fi**.

Tehát mindkét oldal azonos eredményre vezetett.

ÁLLÍTÁS read(delete(t, p), q) = **if** p = q **then** ⊥ **else** read(t, q) **fi**.

BIZONYÍTÁS A tétel bizonyítását strukturális indukcióval próbálhatjuk meg, mivel a kifejezésben kívül nem konstrukciós művelet helyezkedik el.

Alapeset: t = null.

Ekkor read(delete(null, p), q) = read(null, q) = $\perp$, függetlenül attól, hogy p és q egyenlők-e vagy sem.

Indukció: t = assign (t', p', e), ahol feltesszük, hogy az állítás t'-re már igaz. Ekkor

read(delete(assign(t', p', e), p), q) = **if** p' = p **then** read(delete(t', p), q)
else read(assign(delete(t', p), p', e), q) **fi**;

p' = p $\Rightarrow$ read(delete(t', p), q) = **if** p = q **then** $\perp$ **else** read(t', q) **fi**;

p' $\neq$ p $\Rightarrow$ read(assign(delete(t', p), p', e), q) =
if p' = q **then** e **else** read(delete(t', p), q) **fi;**

Az eredmény tehát a következő:

p' = p $\wedge$ p = q $\Rightarrow$ read(delete(t, p), q) = $\perp$;

p' = p $\wedge$ p $\neq$ q $\Rightarrow$ read(delete(t, p), q) = read(t', q);

p' $\neq$ p $\wedge$ p' = q $\Rightarrow$ read(delete(t, p), q) = e;

p' $\neq$ p $\wedge$ p' $\neq$ q $\Rightarrow$ read(delete(t, p), q) = read(t', q);

Ámde

q $\neq$ p' $\Rightarrow$ read(t, q) = **if** q = p' **then** e **else** read(t', q) **fi;**

ÁLLÍTÁS

delete(delete(t, p), p) = delete(t, p).

BIZONYÍTÁS Itt is strukturális indukcióval próbálkozhatunk.

Alapeset: t = null.

a bal oldal:

delete(delete(null, p), p) = delete(null, p) = null,

a jobb oldal:

delete(null, p) = null.

Indukció: t = assign(t', j, a), ahol t'-re az állítás igaz.

a bal oldal:

delete(delete(assign(t', j, a), p), p) = **if** j = p

then delete(delete(t', p), p)

else delete(assign(delete(t', p), j, a), p) **fi;**

Tehát az eredmény

j = p ⇒ delete(delete(assign(t', j, a), p), p) =

delete(delete(t', p), p) = delete(t', p).

j ≠ p ⇒ delete(delete(assign(t', j, a), p), p) =

delete(assign(delete(t', p), j, a), p) =

assign(delete(delete(t', p), p), j, a) =

assign(delete(t', p), j, a).

Látható, hogy a jobb oldal esetén ugyanez a két eset adódik a most kapott eredményekkel.

5.2. A KETTŐS SPECIFIKÁCIÓVAL KAPCSOLATOS ALAPFOGALMAK

Az adattípus osztályspecifikációja tartalmazza az absztrakt adattípus specifikációját és a konkrét adattípus specifikációját. Az elsőt szokás absztrakt specifikációnak, a másodikat pedig konkrét specifikációnak nevezni. Az absztrakt specifikáció elemzésének eredményeként az absztrakt adattípusra rendelkezésünkre áll egy formális specifikáció, amelyet helyesnek fogadhatunk el. Ugyanakkor van egy formálisan megadott konkrét specifikáció is, amelynek a helyességét kívánjuk a már helyesnek elfogadott specifikáció alapján eldönteni. Kettős specifikációról van tehát szó, amelyben az egyiknek a másik szerinti helyességét kell kimutatni. Ehhez szükséges néhány alapfogalom tisztázása.

Adva van tehát egy adattípus absztrakt és konkrét specifikációja:

$d_a = (A, F, E_a)$; $d_c = (C, G, E_c)$;

$A = \{A_0, \ldots, A_n\}$; $C = \{C_0, \ldots, C_m\}$;

$F = \{f_0 \rightarrow A_0, \ldots, f_i: A_j \ldots A_k \rightarrow A_l, \ldots\}$; $G = \{g_0, \ldots, g_i: C_j \ldots C_k \rightarrow C_l, \ldots\}$;

Itt a következő specifikációs módszerek fordulhatnak elő:

Mind "E_a" mind "E_c" háromféle módszerrel lehet megadva.

1. Algebrai axiómákkal adott specifikáció.

   $E_a = \{\ldots f_s(f_c(a)) = h_a(a)\ldots\}$;

   $E_c = \{\ldots g_s(g_c(c))) = h_c(c)\ldots\}$;

2. A külső felületre vonatkozó állításokkal adott specifikáció.

   $E_a = \{\ \ldots, \{pre_f(a)\}\ a' = f(a)\ \{post_f(a, a')\}, \ldots\}$;

   $E_c = \{\ \ldots, \{pre_g(c)\}\ c' = g(c)\{post_g(c, c')\}, \ldots\}$;

3. Procedúrákkal adott specifikáció.

   $E_a = \{\ldots, Q_f, \ldots\}$;

   $E_c = \{\ldots, Q_g, \ldots\}$;

Ezekenek a specifikációknak az egymáshoz való viszonyát a reprezentációs függvény fejezi ki:

$$\varphi: C \rightarrow A$$

Ez általában egy vektorfüggvény:

$$\varphi = (\varphi_0, \ldots, \varphi_n), \text{ ahol}$$

$$\varphi_0: C_0 \rightarrow A_0;$$

$$\varphi_1: C_1 \rightarrow A_1;$$

$$\ldots$$

$$\varphi_n: C_n \rightarrow A_n;$$

Itt azonban $\varphi_1, \ldots, \varphi_n$ általában identitás. Ha az nem értelemzavaró, akkor ezért φ_0 helyett φ -t fogunk írni. Például

$set_a = \{set, elem\}$; $set_c = \{vector \times nat, elem\}$; esetén

φ_0: vector × nat → set; helyett φ-t használunk φ: $set_c \rightarrow set_a$ értelemben is.

Hasonló egyszerűsítés, hogy általános vizsgálat során például $f(\varphi_0(C_0), a_i, a_j)$ helyett $f(\varphi(c))$-t írunk, de konkrét esetekben, rendszerint kiírjuk külön a paramétereket. Például

insert(φ(v, m), e) és nem insert(φ(v, m, e)).

DEFINÍCIÓ Legyenek adva a d_a absztrakt és a d_c konkrét típusspecifikációk, amelyeknek szignatúrája azonos. Legyen megadva továbbá a

$$\varphi: C \rightarrow A, \text{ morfizmus.}$$

A C objektumhalmazt az A egy reprezentációjának nevezzük, ha

$$(\forall a \in A)\ ((\exists c \in C)(a = \varphi(c)).$$

Tehát a definíció azt mondja ki, hogy akkor beszélünk reprezentációról, ha minden absztrakt objektumot reprezentál legalább egy konkrét objektum. Ez természetes is, hiszen például a h = {1, 2, 3} halmazt egyaránt helyesen reprezentálhatja v_1 = 1, 2, 3; v_2 = 2, 3, 1; v_3 = 3, 1, 2;

5.2.1. TÉTEL Legyenek adva a d_a absztrakt és d_c konkrét típusspecifikációk, amelyeknek szignatúrája azonos. Legyen megadva a φ: C → A morfizmus, és legyen $F_c \subset F$ a konstrukciós műveletek halmaza.

Tegyük fel, hogy $\forall f_c \in F_c$ konstrukciós műveletre igaz, hogy

$$a \in A \wedge f_c(a) \in A \wedge c \in C \wedge g_c(c) \in C \wedge a = \varphi(c).$$

Ha

$$(\forall c \in C \wedge \forall f_c \in F_c)(f_c(\varphi(c)) = \varphi(g_c(c))),$$

akkor a C objektumhalmaz az A egy reprezentációja.

BIZONYÍTÁS Strukturális indukcióval járhatunk el.

Alapeset: a = f_0. $f_0 \in A_0$, $g_0 \in C_0$, feltevésünk szerint $f_0 = \varphi(g_0)$.

Tehát $a = f_0$ esetén létezik olyan $c \in C_0$, hogy $a = \varphi(c)$.

Indukció: Legyen $a' = f_c(a)$, ahol feltesszük, hogy

$$a = \varphi(c) \text{ és } c \in C_0.$$

Tehát az $a' = f_c(\varphi(c))$ és a művelettartásra vonatkozó feltevésünk alapján:

$$a' = \varphi(g_c(c)), \text{ ezért}$$

$$c' = g_c(c) \text{ választás mellett } a' = \varphi(c') \text{ és } c' \in C_0.$$

MEGJEGYZÉS A tétel állítása fordítva felhasználható, azaz definiálható vele a reprezentációs függvény.

DEFINÍCIÓ A reprezentációs függvény implicit definíciója:

$$f_0 = \varphi(g_0);$$

$$(\forall f_c \in F_c)(f_c(\varphi(c)) = \varphi(g_c(c)).$$

5.2.1. PÉLDA A halmaz vektorral és mutatóval történő ábrázolása esetén a reprezentációs függvény definíciója:

$$\text{empty} = \varphi(\text{nil}, \text{zerus});$$

$$(\sim\exists i, 1 \le i \le m)(\text{access}(v, i) = e) \Rightarrow$$

$$\text{insert}(\varphi(v, m), e) = \varphi(\text{put}(v, \text{succ}(m), e), \text{succ}(m));$$

$$(\exists i, 1 \le i \le m)(\text{access}(v, i) = e) \Rightarrow \text{insert}(\varphi(v, m), e) = \varphi(v, m).$$

Ha azonos névvel jelöljük a műveleteket a típus osztályának felírásánál, akkor a művelettartást fejezzük ki vele a reprezentáció során. Például

body =

oprs: rep: vector nat $\rightarrow$ set

eqns: $v \in$ vector, $m \in$ nat, $e \in$ elem;

empty = (nil, zerus);

insert(v, m, e) = **if** $(\exists i, 1 \le i \le m)$(access(v, i) = e)

then (v, m) **else** (put(v, succ(m), e), succ(m)) **fi;**

Itt tehát ez a művelettartást fejezi ki:

$$empty_a = \varphi(empty_c)$$

$$insert_a(\varphi(v, m), e) = \varphi(insert_c(v, m, e)),$$

ami általános estben:

$$f_c(\varphi(c)) = \varphi(g_c(c));$$

Itt a c index konstrukciós műveletekre utal.

DEFINÍCIÓ A reprezentációs függvény egy rekurzív (explicit) definíciója.

Tegyük fel, hogy találtunk olyan konstrukciós, szelekciós műveletpárt, amelyre

$$c = g_c(g_s(c)).$$

Ekkor a reprezentációs függvényre a következő módon adhatunk rekurzív definíciót:

$$\varphi(c) = \textbf{if } c = g_0 \textbf{ then } f_0 \textbf{ else } f_c(\varphi(g_s(c))) \textbf{ fi};$$

Például a halmaz esetében, amikor a halmazt egy v vektorral és egy m mutatóval ábrázoljuk, a reprezentációs függvény rekurzív definíciója a következő összefüggéssel kapható:

$$(v, succ(m)) = insert_c(delete_c(v, succ(m)), succ(m), access(v, succ(m))).$$

Így kapjuk, hogy

$$\varphi(v, m) = \textbf{if } m = zerus \textbf{ then } empty \textbf{ else } insert(\varphi(v, prec(m)), m, access(v, m)).$$

Vegyük észre, hogy a reprezentációs függvény most megismert definíciója nem egyértelmű. Ez következik abból, hogy a konstrukciós függvények definíciója sem egyértelmű. Legyen $f_c(a)$ és $f_c'(a)$ két különböző konstrukciós függvény. Konkrét megfelelőjük legyen $g_c(c)$, $g_c'(c)$.

Létezzen két olyan konkrét szelekciós művelet, amelyekre:

$$c = g_c(g_s(c)) \text{ és } c = g_c'(g_s'(c)),$$

akkor

$$\varphi_1(c) = \textbf{if } c = g_0 \textbf{ then } f_0 \textbf{ else } f_c(\varphi(g_s(c))) \textbf{ fi};$$

$$\varphi_2(c) = \textbf{if } c = g_0 \textbf{ then } f_0 \textbf{ else } f_c'(\varphi(g_s'(c))) \textbf{ fi};$$

Ilyenkor jogosan merül fel a kérdés, hogy $\varphi_1(c) = \varphi_2(c)$ fennáll-e. Ennek az egyenlőségnek a bizonyítását strukturális indukcióval végezhetjük el. Ekkor azonban a strukturális indukció a konkrét térben történik

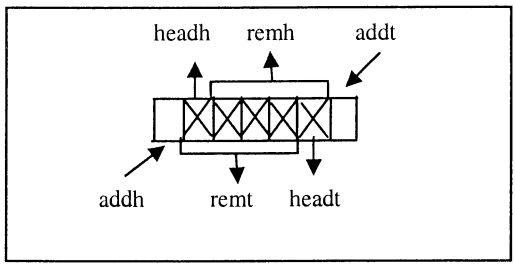
$$(g_0, g_c(c))$$

konkrét konstrukciós műveletek felhasználásával, az egyenlőség axiómát viszont az absztrakt térben alkalmazzuk.

Az egyenlőség axióma az absztrakt térben a következő lesz:

$$\varphi_1(c) = \varphi_2(c) \equiv (\varphi_1(c) = f_0 \wedge \varphi_2(c) = f_0) \vee (\forall f_s \in F_s)(f_s(\varphi_1(c)) = f_s(\varphi_2(c)).$$

5.2.2. PÉLDA Legyen adva a kettős sor absztrakt és konkrét specifikációja. A kettős sor absztrakt modelljét a következő ábra szemlélteti:



addh: a kettős sor elé egy új elemet helyez el, és az így létrehozott kettős sort adja meg eredményül;

addt: a kettős sor mögé egy új elemet helyez el, és az így létrehozott kettős sort adja meg;

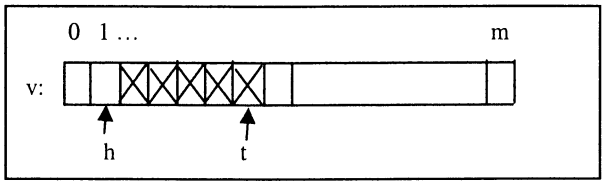
headh: kiolvassa a kettős sor elején álló elemet;

remh: a kettős sor elején álló elemétől megfosztott kettős sort adja meg;

headt: kiolvassa a kettős sor végén álló elemet;

remt: a kettős sor végén álló elemétől megfosztott kettős sort adja meg.

Egy lehetséges ábrázolás egy v vektorral és két, (h, t) mutatóval történhet:



Az informális specifikáció alapján a formális specifikáció — algebrai specifikációt használva — a következő lesz:

$$\text{dque} = (\{\text{dque, elem}\}, \{\text{init}:\rightarrow \text{dque, addh: dque elem} \rightarrow \text{dque,}$$

$$\text{headh: dque} \rightarrow \text{elem, remh: dque} \rightarrow \text{dque, addt: dque elem} \rightarrow \text{dque,}$$

$$\text{headt: dque} \rightarrow \text{elem, remt: dque} \rightarrow \text{dque, length: dque} \rightarrow \text{natbool, } \perp: \rightarrow \text{elem}\},$$

$$\{\text{headh(init)} = \perp, \text{remh(init)} = \text{init, length(init)} = 0,$$

$$\text{addt(init, e)} = \text{addh(init, e)}, \text{headt(init)} = \perp, \text{remt(init)} = \text{init,}$$

$$\text{length(addh(q, e)} = \text{length(q)} + 1, \text{addt(addh(q, e), e')} = \text{addh(addt(q, e'), e),}$$

$$\text{headh(addh(q, e))} = \text{e, remh(addh(q, e))} = \text{q,}$$

$$\text{headt(addh(q, e))} = \textbf{if } \text{q} = \text{init } \textbf{then } \text{e } \textbf{else } \text{headt(q)} \textbf{ fi},$$

$$\text{remt(addh(q, e))} = \textbf{if } \text{q} = \text{init } \textbf{then } \text{init } \textbf{else } \text{remt(q)} \textbf{ fi}\})$$

A konkrét konstrukciós műveletek az adott ábrázolás mellett a következők lehetnek:

$$\text{init}_c = (\text{nil}, 0, 0),$$

$$\text{addh}_c(\text{v, h, t}) = (\text{put(v, h, e)}, \text{h} \ominus 1, \text{t}),$$

itt a rövidség kedvéért $\ominus$ moduló m szerinti kivonást jelent.

Egy másik megoldás a konstrukciós műveletekre:

$$\text{init}_c = (\text{nil}, 0, 0),$$

$$\text{addt}_c(\text{v, h, t}) = (\text{put(v, t} \oplus 1, \text{e)}, \text{h, t} \oplus 1),$$

ahol $\oplus$ moduló m szerinti összeadást jelent.

Ezeknek a reprezentációs függvényeknek a képzéséhez rendre a következő szelekciós függvények társíthatók:

$$remh_c(v, h, t) = (v, h \oplus 1, t)$$

$$head_c(v, h, t) = access(v, h \oplus 1),$$

illetve

$$remt_c(v, h, t) = (v, h, t \ominus 1)$$

$$headt_c(v, h, t) = access(v, t)$$

A két reprezentációs függvény ezen társítások alapján a következő lesz:

$\varphi_h(v, h, t) =$ **if** $h = t$ **then** init **else** $addh(\varphi_h(v, h \oplus 1, t), access(v, h \oplus 1)$ **fi**;

$\varphi_t(v, h, t) =$ **if** $h = t$ **then** init **else** $addt(\varphi_t(v, h, t \ominus 1), access(v, t \ominus 1)$ **fi**.

Most tehát van két reprezentációs függvényünk. A kérdés az, hogy ezek ugyanazt a leképezést valósítják-e meg.

5.2.2. TÉTEL $\varphi_h(v, h, t) = \varphi_t(v, h, t)$

BIZONYÍTÁS Egyenlőség axióma:

$$q_1, q_2 \in dque.$$

$q_1 = q_2 \equiv (q_1 = init \wedge q_2 = init) \vee (remh(q_1) = remh(q_2) \wedge headh(q_1) = headh(q_2))$

Teljes indukciót alkalmazhatunk a konkrét térben. Vegyük azonban észre, hogy most ez az indukció a kettős sor konkrét hossza szerinti indukcióra redukálható:

MEGJEGYZÉS: $0 \le h < m \wedge 0 \le t < m \wedge h = t \wedge (\forall v \in vector) \Rightarrow init_c = (v, h, t)$.

Másrészt

$$length_c(init_c) = 0,$$

$length_c(add_c(v, h, t, e)) = length(v, h, t) + 1$, ezért

$$length_c(v, h, t) = t \ominus h.$$

A strukturális indukciót tehát a konkrét hossz alapján végezhetjük el.

Alapeset: Legyen $t = h$.

$$\varphi_h(v, h, t) = \text{init}, \varphi_t(v, h, t,) = \text{init}.$$

Indukció: Legyen $t \ominus h > t' \ominus h' \geq 0$, ahol minden (v, h', t') konkrét kettős sor objektumra igaz az állítás, és bizonyítsuk minden (v, h, t)-re is.

Alkalmazzuk mindkét reprezentációs függvényre az egyenlőség axiómát.

$$q_1 = \text{addh}(\varphi_h(v, h \oplus 1, t), \text{access}(v, h \oplus 1)),$$

$$q_2 = \text{addt}(\varphi_t(v, h, t \ominus 1), \text{access}(v, t)),$$

$$\text{headh}(q_1) = \text{headh}(\text{addh}(\varphi_h(v, h \oplus 1, t), \text{access}(v, h \oplus 1))) = \text{access}(v, h \oplus 1)$$

$$\text{headh}(q_2) = \text{headh}(\text{addt}(\varphi_t(v, h, t \ominus 1), \text{access}(v, t))) =$$

$$\textbf{if } \varphi_t(v, h, t \ominus 1) = \text{init } \textbf{then } \text{access}(v, t) \textbf{ else } \text{headh}(\varphi_t(v, h, t \ominus 1))$$

$$h = t \ominus 1 \Rightarrow h \oplus 1 = t \wedge \text{access}(v, t) = \text{access}(v, h \oplus 1).$$

$$h \neq t \ominus 1 \Rightarrow \text{headh}(\varphi_t(v, h, t \ominus 1)) = \text{headh}(\varphi_h(v, h, t \ominus 1)) =$$

$$\text{headh}(\text{addh}(\varphi_h(v, h \oplus 1, t \ominus 1), \text{access}(v, h \oplus 1)))) = \text{access}(v, h \oplus 1).$$

$$\text{remh}(q_1) = \text{remh}(\text{addh}(\varphi_h(v, h \oplus 1, t), \text{access}(v, h \oplus 1)) = \varphi_h(v, h \oplus 1, t);$$

$$\text{remh}(q_2) = \text{remh}(\text{addt}(\varphi_t(v, h, t \ominus 1), \text{access}(v, t))) =$$

$$\textbf{if } \varphi_t(v, h, t \ominus 1) = \text{init } \textbf{then } \text{init } \textbf{else } \text{addt}(\text{remh}(\varphi_t(v, h, t \ominus 1), \text{access}(v, t)) \textbf{ fi;}$$

$$h = t \ominus 1 \Rightarrow \text{rem}(q_1) = \text{init} \wedge \text{rem}(q_2) = \text{init};$$

$$h \neq t \ominus 1 \Rightarrow \text{remh}(q_2) = \text{addt}(\text{remh}(\varphi_h(v, h, t \ominus 1), \text{access}(v, t)) =$$

$$\text{addt}(\text{remh}(\text{addh}(\varphi_h(v, h \oplus 1, t \ominus 1), \text{access}(v, h \oplus 1))), \text{access}(v, t)) =$$

$$\text{addt}(\varphi_h(v, h \oplus 1, t \ominus 1), \text{access}(v, t)) =$$

$$\text{addt}(\varphi_t(v, h \oplus 1, t \ominus 1), \text{access}(v, t)) = \varphi_t(v, h \oplus 1, t).$$

A két reprezentációs függvény egyenlőségét tehát bebizonyítottuk.

5.3. AZ IMPLEMENTÁCIÓ HELYESSÉGE

Az előzőekben módszert adtunk a reprezentáció helyességének eldöntésére. Az absztrakt adattípus megvalósítása a reprezentációt és az implementációt foglalja magába. Ha el akarjuk dönteni, hogy jó-e az implementáció, akkor előbb definiálni kell, hogy mit értünk azon, hogy a konkrét specifikáció az absztrakt specifikáció szerint helyes. Erre a következő definíciót adjuk:

DEFINÍCIÓ Legyen adva a $d_a = (A, F, E_a)$ absztrakt specifikáció és a $d_c = (C, G, E_c)$ konkrét specifikáció, amelyeknek a szignatúrája azonos. Legyen továbbá megadva a $\varphi: C \rightarrow A$ morfizmus.

Ha

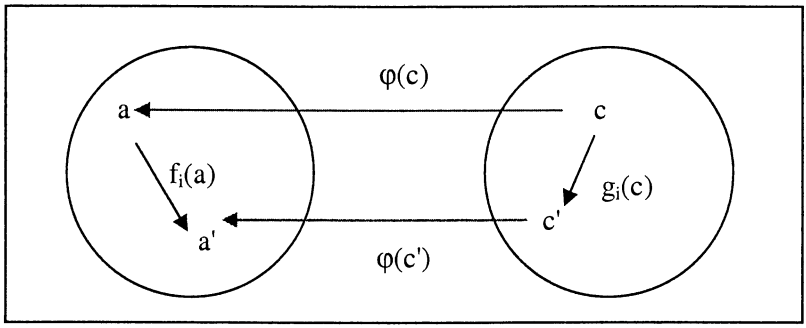
1. C az A egy reprezentációja az adott φ mellett;
2. $(\forall f_i \in F)(c \in C \wedge \varphi(c) \in A \wedge f_i(\varphi(c))$ értelmezve van, akkor $g_i(c)$ is értelmezve van;
3. $(\forall f_i \in F)(c \in C \wedge c' = g_i(c) \wedge c' \in C \wedge \varphi(c) \in A \wedge \varphi(c') \in A$, akkor $f_i(\varphi(c)) = \varphi(g_i(c))$,

mindezek igazak, akkor d_c a d_a szerint helyes.

MEGJEGYZÉS: Ha $p = f_i(a)$ és $q = g_i(c)$ ahol p, q paraméterek, akkor

$f_i(\varphi(c)) = \varphi(g_i(c))$ helyébe $p = q$ lép.

Vegyük észre, hogy az implementáció helyességének ez a definíciója azt fogalmazza meg, hogy minden $g_i(c)$ konkrét művelet az $f_i(a)$ absztrakt művelet leképezését szimulálja a konkrét térben. Ezt szemlélteti a következő ábra morfizmusdiagramja.



5.3.1. ÁLLÍTÁS Legyen adva $d_a = (A, F, E_a)$, $d_c = (C, G, E_c)$, φ: $C \rightarrow A$, és legyen d_c a d_a szerint helyes. Adott továbbá a P_a absztrakt program, amelynek programfüggvénye $p_a(a)$. Állítsuk elő a P_c konkrét programot a P_a absztrakt programból úgy, hogy

$\forall a \in A$ helyére a megfelelő $c \in C$-t tesszük,

$\forall f_i \in F$ helyére a megfelelő $g_i \in G$-t tesszük.

Ha a konkrét program programfüggvénye $p_c(c)$, és a programok indulásakor $a_0 = \varphi(c_0)$, akkor

$$p_a(\varphi(c_0)) = \varphi(p_c(c_0)).$$

AZ ÁLLÍTÁS SZEMLÉLTETÉSE Legyen

$$d_a = (\{\text{verem, elem}\}, \{\text{create, push, pop, top, } \ldots\}, E_a),$$

$$d_c = (\{\text{vector} \times \text{nat, elem}\}, \{(\text{nil}, 0), (\text{put}(v, i, e), i + 1), \ldots\}, E_c),$$

$$\varphi\text{: vector nat} \rightarrow \text{verem},$$

$$\varphi(v, m) = \textbf{if } m = 0 \textbf{ then create else } \text{push}(\varphi(v, m - 1), \text{access}(v, m)) \textbf{ fi};$$

$x \in$ verem, $y \in$ verem, v, w$\in$ vector, i, k, m$\in$ nat, $x = \varphi(v, m)$.

Absztrakt program:	Konkrét program:
y ← create;	(w, k) ← (new, 0);
while x ≠ create **do**	**while** m ≠ 0 **do**
y ← push(y, top(x));	(w, k) ← (put(w, k + 1, access(v, m)), k + 1);
x ← pop(x);	(v, m) ← (v, m - 1);
od;	**od**;

AZ ÁLLÍTÁS BIZONYÍTÁSA Az adattípus programban szereplő műveleteinek száma szerinti teljes indukcióval történhet a bizonyítás.

Alapeset: $a_0 = \varphi(c_0)$, ami a feltevésünk. Például create = φ(nil, 0);

Indukció: a k + 1-edik művelet előtt igaz az (a, c) részeredményekre, hogy

$$a = \varphi(c).$$

Például y = φ(w, k).

A k + 1-edik műveletpár legyen (f(a), g(c)).

Akkor feltevésünk szerint $f(\varphi(c)) = \varphi(g(c))$ és az utolsó lépés eredménye így:

$$a' = p_a(a_0) \text{ és } c' = p_c(c_0).$$

Ezért

$$a' = \varphi(c'),$$

azaz

$$p_a(a_0) = \varphi(p_c(c_0)),\ p_a(\varphi(c_0)) = \varphi(p_c(c_0)).$$

Ezzel a bizonyítás teljes.

5.4. A MEGVALÓSÍTÁSOK ELEMZÉSE

5.4.1. ÁLLÍTÁS Az algebrai specifikáció elő- és utófeltételes formára hozható.

BIZONYÍTÁS Legyen adott

$$E_a = \{\ldots, \alpha(a) \Rightarrow f_s(f_c(a)) = h(a), \ldots, \sim I_a(f_c(a)) \Rightarrow f_c(a) = \text{"undefined"}\},$$

ahol

$$I_a(a) = 0 \leq attr(a) \leq n.$$

Átírás: $\forall f_s \in F_s$ esetén:

$$\{\alpha(a) \wedge b = f_c(a)\}\ b' = f_s(b)\ \{b' = h(a)\}$$

f_0-ra:

$$\{\text{"true"}\}\ a = f_0\ \{I_a(a)\}$$

f_c-re:

$$\{I_a(f_c(a))\}\ b = f_c(a)\ \{I_a(b) \wedge b = f_c(a)\}$$

5.4.1. PÉLDA Hozzuk a halmaz szemantikáját meghatározó algebrai axiómákat elő- és utófeltételes formára. A halmaz esetén:

$$I_a(s): 0 \leq size(s) \leq n;$$

Például nem konstrukciós műveletre

$$has(insert(s, e), e') = \textbf{if } e = e' \textbf{ then } \text{"true"} \textbf{ else } has(s, e') \textbf{ fi;}$$

az axióma átírása:

$$\{e = e' \wedge a = insert(s, e)\}\ b = has(a, e')\ \{b = \text{"true"}\};$$

$$\{e \neq e' \wedge a = insert(s, e)\}\ b = has(a, e')\ \{b = has(s, e')\}.$$

Konstrukciós műveletekre

$$empty \in \{s \mid 0 \leq size(s) \leq n\},$$

$$s' \in \{s \mid 0 \leq size(s) \leq n\} \Rightarrow insert(s', e) \in \{s \mid 0 \leq size(s) \leq n\}.$$

Másképpen:

$$size(s, e) > n \Rightarrow size(s, e) = \text{"undefined"}.$$

Az átírás eredménye

$$\{\text{"true"}\}\ s = \text{empty}\ \{0 \le \text{size}(s) \le n\}$$

$$\{0 \le \text{size}(\text{insert}(s, e)) \le n\}\ s' = \text{insert}(s, e)\ \{0 \le \text{size}(s') \le n \wedge s' = \text{insert}(s, e)\}$$

5.4.2. ÁLLÍTÁS (A külső felület specifikációjával adott konkrét specifikáció absztrakt specifikáció szerinti helyességének egy elégséges feltétele.)

Legyen adott $d_a = (A, F, E_a)$ és $d_c = (C, G, E_c)$, ahol mind E_a, mind E_c elő- és utófeltételes formában van megadva. Azaz

$$E_a = \{\{\text{"true"}\}\ a = f_0\ \{post_{f_0}(a)\}, \ldots, \{pre_{f_i}(a)\}\ a' = f_i(a)\ \{post_{f_i}(a, a')\}, \ldots\},$$

$$E_c = \{\{\text{"true"}\}\ c = g_0\ \{post_{g_i}(c)\}, \ldots, \{pre_{g_i}(c)\}\ c' = g_i(c)\ \{post_{g_i}(c, c')\}, \ldots\}.$$

Legyen továbbá

$$A_0 = \{a \mid I_a(a)\},\ C_0 = \{c \mid I_c(c)\},\ \varphi\colon C \to A.$$

Ha bebizonyítjuk, hogy

1.) $I_c(c) \Rightarrow I_a(\varphi(c))$;

2.) $post_{g_0}(c) \Rightarrow I_c(c)$;

3.) $post_{g_0}(c) \Rightarrow post_{f_0}(\varphi(c))$;

4.) $I_c(c) \wedge pre_{f_i}(\varphi(c)) \Rightarrow pre_{g_i}(c)$;

5.) $I_c(c) \wedge pre_{f_i}(\varphi(c)) \wedge post_{g_i}(c, c') \wedge I_c(c') \Rightarrow post_{f_i}(\varphi(c), \varphi(c'))$;

6.) $I_c(c') \wedge pre_{g_i}(c') \wedge post_{g_i}(c, c') \Rightarrow I_c(c)$,

akkor a d_c konkrét specifikáció a d_a absztrakt specifikáció szerint helyes.

BIZONYÍTÁS

a. Megmutatjuk, hogy C az A egy reprezentációja.

- $a = f_0$. 2. és 1. szerint $g_0 \in C \wedge \varphi(g_0) \in A$. 3. szerint $\varphi(g_0) = f_0$.
- 4. szerint ha $f_c \in F_c$, $f_c(\varphi(c))$ értelmezve van, akkor $g_c(c)$ is értelmezve van.
- 5. szerint ekkor $\varphi(c') = f_c(\varphi(c)) \wedge c' = g_c(c)$, azaz $f_c(\varphi(c)) = \varphi(g_c(c))$

b. Morfizmusdiagram szerinti kapcsolat

- 4. szerint ha $\forall f_i \in F$ esetén, ha $f_i(\varphi(c))$ értelmezve van, akkor $g_i(c)$ is értelmezve van.
- 5. és 1. szerint $c' = g_i(c) \wedge I_c(c') \wedge I_a(\varphi(c')) \wedge \varphi(c') = f_i(\varphi(c))$, azaz $(\forall f \in F)(\varphi(g(c)) = f(\varphi(g(c))))$.

Ezzel a bizonyítás teljes.

Összefoglalás

Reprezentációelemzés során a következőket kell megvizsgálni:

- $\varphi_1(c) = \varphi_2(c)$?
- $length_c(c) = lenght_a(\varphi(c))$?
- $c_1 = c_2 \Rightarrow \varphi(c_1) = \varphi(c_2)$?
- $I_c(c) \Rightarrow I_a(\varphi(c))$?
- Ha $attr_c(c) = attr_a(\varphi(c))$ és $I_c(c)$: $0 \leq attr_c(c) \leq n$ és $I_a(\varphi(c))$: $0 \leq attr_a(\varphi(c)) \leq n$, akkor az $I_c(c) \Rightarrow I_a(\varphi(c))$ állítás triviális.

Implementációelemzés során a következő kérdésekre keressük a választ:

- $post_{g_0}(c) \Rightarrow post_{f_0}(\varphi(c))$?
- $I_c(c) \wedge pre_{f_i}(\varphi(c)) \Rightarrow pre_{g_i}(c)$?
- $I_c(c) \wedge pre_{f_i}(\varphi(c)) \wedge post_{g_i}(c, c') \wedge I_c(c') \Rightarrow post_{f_i}(\varphi(c), \varphi(c'))$?

5.4.2. PÉLDA A halmaz osztály elemzése.

Konkrét invariáns: $I_c(v, m)$: $0 \leq m \leq n$

Absztrakt invariáns: $I_a(s)$: $0 \leq size(s) \leq m$

Legyen a reprezentációs függvény a következő:

$\varphi(v, m)$ = **if** $m = 0$ **then** empty **else** insert($\varphi(v, m - 1)$, access(v, m)) **fi;**

A has(s, e) jelentésének absztrakt axiómája átírva:

$$\{e = e' \wedge a = insert(s, e)\}\ b = has(a, e')\ \{b = \text{"true"}\}$$

$$\{e \neq e' \wedge a = insert(s, e)\}\ b = has(a, e')\ \{b = has(s, e')\};$$

A $has_c(v, m, e)$ jelentésének konkrét specifikációja:

$$\{0 \leq m \leq n\}\ b' = has_c(v, m, e')\ \{b' = (\exists i, 1 \leq i \leq m)(access(v, i) = e')\}.$$

Az insert(s, e) jelentésének absztrakt leírása átírva:

$$\{\text{"true"}\}\ s = empty\ \{0 \leq size(s) \leq n\};$$

$$\{0 \leq size(insert(s, e)) \leq n\}\ s' = insert(s, e)\ \{0 \leq size(s') \leq n \wedge s' = insert(s, e)\}.$$

Az $insert_c(v, m, e)$ jelentésének konkrét specifikációja:

$$\{\text{"true"}\}\ (v, m) = empty_c\ \{m = 0\};$$

$$\{0 \leq m < n \vee (m = n \wedge (\exists j, 1 \leq j \leq m)(access(v, j) = e)\}$$

$$(v', m') = insert_c(v, m, e)$$

$$\{[(\exists j, 1 \leq j \leq m)(access(v, j) = e) \wedge (v' = v \wedge m' = m)] \vee$$

$$[(\sim\exists j, 1 \leq j < m)(access(v, j) = e) \wedge (v' = put(v, m + 1, e) \wedge m' = m + 1)].$$

BIZONYÍTSUK BE A $has_c(v, m, e)$ HELYESSÉGÉT:

1., vagyis $I_c(c) = I_a(\varphi(c))$ teljesül, azaz

$$0 \leq m \leq n \Rightarrow 0 \leq size(\varphi(v, m)) \leq m, \text{ mivel } m = size(\varphi(v, m)).$$

Ez — a reprezentációs függvény alapján nyilvánvaló igazság — m szerinti teljes indukcióval bizonyítható.

4., vagyis $I_c(c) \wedge pre_{f_i}(\varphi(c)) \Rightarrow pre_{g_i}(c)$ teljesül, azaz

$$0 \leq m \leq n \wedge e = e' \wedge \varphi(v, m) = insert(s, e) \Rightarrow 1 \leq m \leq n.$$

$$0 \leq m \leq n \wedge e \neq e' \wedge \varphi(v, m) = insert(s, e) \Rightarrow 1 \leq m \leq n.$$

Bizonyítás $length_c(v, m) = length(\varphi(v, m)) = length(s) + 1 > 0.$

5., vagyis $I_c(c) \wedge pre_{f_i}(\varphi(c)) \wedge post_{g_i}(c, c') \wedge I_c(c') \Rightarrow post_{f_i}(\varphi(c), \varphi(c'))$ teljesül, azaz

a) $0 \le m \le n \wedge e = e' \wedge \varphi(v, m) = insert(s, e) \wedge e = e' \wedge$

$(b' = (\exists i, 1 \le i \le m)(access(v, i) = e')) \Rightarrow b = b'.$

Bizonyítás $b = has(\varphi(v, m), e) = has(insert(s, e), e) = \text{"true"}.$

b) $0 \le m \le n \wedge e \ne e' \wedge \varphi(v, m) = insert(s, e) \wedge b' =$

$(\exists i, 1 \le i \le m)(access(v, i) = e') \Rightarrow b = b'.$

A bizonyítás a fentihez hasonlóan történhet.

ADJUK MEG AZ $insert_c(v, m, e)$ **HELYESSÉGÉNEK BIZONYÍTÁSÁHOZ SZÜKSÉGES TÉTELEKET:**

4. $0 \le m \le n \wedge 0 \le size(insert(\varphi(v, m), e)) \le n$

$\Rightarrow 0 \le m < n \vee (m = n \wedge (\exists j, 1 \le j \le m)(access(v, j) = e)).$

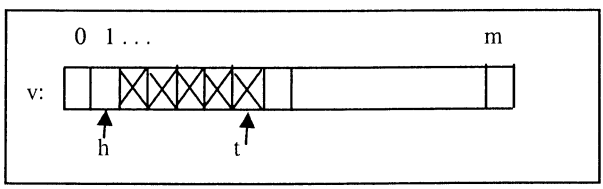
5. $(0 \le m \le n \wedge 0 \le size(insert(\varphi(v, m), e)) \le n) \wedge$

$([(\exists j, 1 \le j \le m)(access(v, j) = e) \wedge (v' = v \wedge m' = m)] \vee$

$[(\sim\exists j, 1 \le j < m)(access(v, j) = e) \wedge (v' = put(v, m + 1, e) \wedge m' = m + 1)]) \Rightarrow$

$0 \le size(\varphi(v', m')) \le n \wedge \varphi(v', m') = insert(\varphi(v, m), e).$

Válasszuk példaként a kettős sort a korábbi ábrázolás mellett:



dque = ({dque, elem}, {init:→ dque, addh: dque elem → dque,

headh: dque → elem, remh: dque → dque, addt: dque elem → dque,

headt: dque → elem, remt: dque → dque, length: dque → natbool,

⊥: → elem},

{headh(init) = ⊥, remh(init) = init, length(init) = 0,

addt(init, e) = addh(init, e), headt(init) = $\perp$,

remt(init) = init, length(addh(q, e)) = length(q) + 1,

addt(addh(q, e), e') = addh(addt(q, e'), e),

headh(addh(q, e)) = e, remh(addh(q, e)) = q,

headt(addh(q, e)) = **if** q = init **then** e **else** headt(q) **fi**,

remt(addh(q, e)) = **if** q = init **then** init **else** remt(q) **fi**})

Az adott ábrázolás mellett a konkrét konstrukciós műveletek:

$init_c$ = (nil, 0, 0),

$addh_c(v, h, t, e)$ = (put(v, h, e), h $\ominus$ 1, t);

a nem konstrukciós műveletek:

$n > t \ominus h \geq 0 \Rightarrow addt_c(v, h, t, e) = (put(v, t \oplus 1, e), h, t \oplus 1)$

$n \geq t \ominus h > 0 \Rightarrow remt_c(v, h, t) = (v, h, t \ominus 1)$

$n \geq t \ominus h > 0 \Rightarrow headt_c(v, h, t) = access(v, t)$

$n \geq t \ominus h > 0 \Rightarrow remh_c(v, h, t) = (v, h \oplus 1, t)$

$n \geq t \ominus h > 0 \Rightarrow headh_c(v, h, t) = access(v, h \oplus 1)$

$length_c(v, h, t) = t \ominus h.$

A két reprezentációs függvény:

$\varphi_h(v, h, t)$ = **if** h = t **then** init

else addh(φ_h(v, h $\oplus$ 1, t), access(v, h $\oplus$ 1)) **fi**;

$\varphi_t(v, h, t)$ = **if** h = t **then** init

else addt(φ_t(v, h, t $\ominus$ 1), access(v, t)) **fi**.

Egyenlőség axióma az absztrakt térben:

$q_1, q_2 \in$ dque,

$q_1 = q_2 \equiv (q_1 = \text{init} \wedge q_2 = \text{init}) \vee (\text{remh}(q_1) = \text{remh}(q_2) \wedge$

$\text{headh}(q_1) = \text{headh}(q_2))$.

Egyenlőség axióma a konkrét térben:

v, v'∈ vector; h, h', t, t'∈ natbool;

$(v, h, t) = (v', h', t') \equiv (h = t \wedge h' = t') \vee (t \ominus h = t' \ominus h' \wedge$

$(\forall j, 1 \leq j \leq t \ominus h)(\text{access}(v, h \oplus j) = \text{access}(v', h' \oplus j)))$.

5.4.3. PÉLDA A reprezentáció elemzése a kettős sor esetében. A reprezentációs függvények egyenlőségét a $\varphi_h(v, h, t) = \varphi_t(v, h, t)$ esetében már bebizonyítottuk. A továbbiakban φ_h és φ_t helyett φ-t írhatunk. Igazak a következő állítások a kettős sorra és megvalósítására.

ÁLLÍTÁS $\text{length}_c(v, h, t) = \text{length}_a(\varphi(v, h, t))$

BIZONYÍTÁS:

- $h = t \Rightarrow \varphi(v, h, t) = \text{init}$; ezért $\text{length}_c(v, h, t) = t \ominus h = 0$

  és $\text{length}_a(\varphi(v, h, t)) = 0$.

- $t \ominus h > 0$, és $t' = t$, $h' = h \oplus 1$, $v' = v$ mellett:

  $\text{length}_c(v', h', t') = \text{lenght}_a(\varphi(v', h', t'))$.

  $\text{length}_c(v, h, t) = t \ominus h = t' \ominus h' \oplus 1 = \text{length}_c(v', h', t') + 1$.

  $\text{length}_a(\varphi(v, h, t)) = \text{length}_a(\text{addh}(\varphi(v, h \oplus 1, t), \text{access}(v, h \oplus t))) =$

  $\text{length}_a(\varphi(v, h \oplus 1, t)) + 1 = \text{length}_a(\varphi(v', h', t')) + 1$.

ÁLLÍTÁS: $(v, h, t) = (v, h', t') \Rightarrow \varphi(v, h, t) = \varphi(v, h', t')$, azaz a konkrét kettős sor egyenlőségéből következik az absztrakt kettős sor egyenlősége is.

BIZONYÍTÁS:

- $h = t \wedge h' = t' \Rightarrow \varphi(v, h, t) = \varphi(v, h', t') = init.$
- $t \ominus h = t' \ominus h' > 0$ és feltesszük, hogy $t_1 \ominus h_1 = t_2 \ominus h_2$ és

  $t_2 \ominus h_2 < t \ominus h$ esetén az állítás igaz.

  Ekkor $\varphi(v, h, t) = addh(\varphi(v, h \oplus 1, t),$

  $access(v, h \oplus 1)) = addh(\varphi(v, h' \oplus 1, t'), access(v, h' \oplus 1)) = \varphi(v, h', t').$

Absztrakt invariáns:

$$I_a(q): 0 \leq length(q) \leq n.$$

Konkrét invariáns:

$$I_c(v, h, t): 0 \leq t \ominus h \leq n.$$

ÁLLÍTÁS $remh_c(v, h, t)$ helyes.

BIZONYÍTÁS Az absztrakt specifikáció átírása:

$$\{q' = addh(q, e)\}\ q'' = remh(q')\ \{q'' = q\}.$$

A konkrét specifikáció átírása:

$$\{n \geq t \ominus h > 0\}\ (v', h', t') = remh_c(v, h, t)\ \{v' = v \wedge h' = h \oplus 1 \wedge t' = t\}.$$

A bizonyítandó tételek:

4. $0 \leq t \ominus h \leq n \wedge \varphi(v, h, t) = addh(q, e) \Rightarrow 0 < t \ominus h \leq n;$

   Bizonyítás: $t \ominus h = length_c(v, h, t) = length(\varphi(v, h, t)) =$

   $length(addh(q, e)) = length(q) + 1 > 0.$

5. $0 \leq t \ominus h \leq n \wedge \varphi(v, h, t) = addh(q, e) \wedge (v' = v \wedge h' = h \oplus 1 \wedge t' = t) \wedge$

   $0 \leq t' \ominus h' \leq n \wedge 0 \leq length(\varphi(v, h, t)) \leq n \Rightarrow \varphi(v', h', t') = q.$

   Bizonyítás: $addh(q, e) = \varphi(v, h, t) =$

   $addh(\varphi(v, h \oplus 1, t), access(v, h \oplus 1));$

azaz

$$q = \varphi(v, h \oplus 1, t) = \varphi(v', h', t').$$

ÁLLÍTÁS $headh_c(v, h, t)$ helyes.

BIZONYÍTÁS Az absztrakt specifikáció átírása:

$$\{q' = addh(q, e)\}\ e' = headh(q')\ \{e' = e\}.$$

A konkrét specifikáció átírása:

$$\{n \geq t \ominus h > 0\}\ a = readh(v, h, t)\ \{a = access(v, h \oplus 1)\}.$$

A bizonyítandó tételek:

4. $0 \leq t \ominus h \leq n \wedge \varphi(v, h, t) = addh(q, e) \Rightarrow 0 < t \ominus h \leq n$;

5. $0 \leq t \ominus h \leq n \wedge \varphi(v, h, t) = addh(q, e) \wedge a = access(v, h \oplus 1) \Rightarrow a = e$.

   Bizonyítás:

$$addh(q, e) = \varphi(v, h, t) = addh(\varphi(v, h \oplus 1, t), access(v, h \oplus 1));$$

$$azaz\ e = access(v, h \oplus 1)) = a;$$

5.5. A KETTŐS SPECIFIKÁCIÓ HELYESSÉGÉNEK KÉTFÉLE MEGKÖZELÍTÉSE

A konkrét adattípus specifikációjának helyességét a szimuláció alapján definiáltuk. Ez azt jelentette, hogy a konkrét műveletek a konkrét térben az absztrakt műveletek leképezését szimulálják. Másik lehetséges megközelítése lehet az, hogy az absztrakt és a konkrét műveletek jelentése azonos. A következőkben ezt a két megközelítést pontosan megfogalmazzuk, és ezek ekvivalenciáját bebizonyítjuk.

Legyen $d_a = (A, F, E_a)$; absztrakt és $d_c = (C, G, E_c)$; konkrét specifikáció.

$$A = \{A_0, A_1, \ldots, A_n\}; \qquad F = \{f_0, f_1, \ldots, f_m\};$$

$$C = \{C_0, C_1, \ldots, C_n\}; \qquad G = \{g_0, g_1, \ldots, g_m\};$$

$E_a = \{\ldots, e_{a_j}(\ldots, f_j(a), \ldots; a), \ldots\}$; $E_c = \{\ldots, e_{c_j}(\ldots, g_j(c), \ldots; c), \ldots\}$;

ahol e_{a_j}, e_{c_j} az absztrakt, ill. konkrét szemantikai axiómákat jelöli.

Legyen megadva továbbá a reprezentációs függvény

$$\varphi: C \rightarrow A.$$

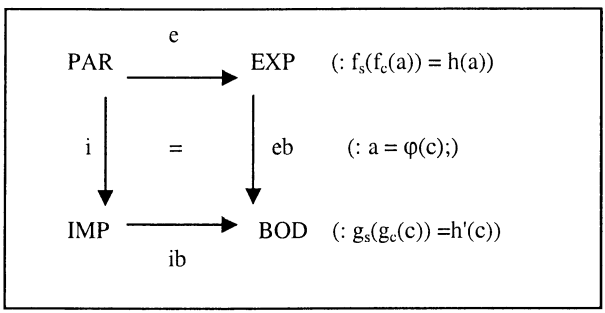
$$f_0 = \varphi(g_0),$$

$$(\forall f_c \in F_c)(\forall c \in C)(f_c(\varphi(c)) = \varphi(g_c(c))).$$

A helyesség definíciójának szimuláción alapuló lényege a következő:

$$(\forall f_s \in F_s)(\forall c \in C)(f_s(\varphi(c)) = \varphi(g_s(c))),$$

Ezzel szemben az, ahogyan a típusosztály morfizmusdiagramját definiáltuk, azonos szemantikát jelentett. Mi ugyanis azt követeltük meg, hogy a szemantikai axiómák az exp részből a reprezentációnak megfelelően változatlan formában másolódjanak át a body részbe. Ezt szemlélteti a következő ábra:



Általános formában ez a következőképpen írható:

$$e_{a_j}(\ldots, f_j(\varphi(c)), \ldots; \varphi(c)) \equiv e_{c_j}(\ldots, g_j(c), \ldots; c).$$

Az azonos jelentés azonos specifikációs módszereket használva a következő:

- "Algebrai–algebrai" esetnél az alábbi speciális axiómára a következő azonosságot kapjuk:

$$f_s(f_c(\varphi(c)) = f_c(f_s(\varphi(c))) \equiv g_s(g_c(c)) = g_c(g_s(c)).$$

- "Külső felület–külső felület" esetnél a következő azonosság fejezi ki az azonos jelentést:

$$(\text{pre}_{f_i}(\varphi(c)) \wedge \text{post}_{f_i}(\varphi(c), \varphi(c'))) \equiv (\text{pre}_{g_i}(c) \wedge \text{post}_{g_i}(c, c')).$$

5.5.1. PÉLDA Tekintsük a halmaz algebrai specifikációját. Vizsgáljuk az absztrakt specifikáció következő axiómáját:

$$\text{delete}(\text{insert}(s, e), e) = \text{delete}(s, e).$$

A vektorral és egy mutatóval történő ábrázolás mellett egy lehetséges reprezentációs függvény:

$$\varphi(v, m) = \textbf{if } m = \text{zerus } \textbf{then } \text{empty}$$

$$\textbf{else } \text{insert}(\varphi(v, \text{prec}(m)), \text{access}(v, m)) \textbf{ fi};$$

Az adott ábrázolás mellett a szimuláción alapuló helyességet a következő egyenletek fejezik ki:

$$\text{empty} = \varphi(\text{empty}_c); \quad \text{insert}(\varphi(v, m), e) = \varphi(\text{insert}_c(v, m, e));$$

$$\text{delete}(\varphi(v, m), e) = \varphi(\text{delete}_c(v, m, e));$$

Kérdés, hogy a konkrét műveletek kielégítik-e az absztrakt axiómákat? Nézzük a delete_c esetét. Kérdés tehát az, hogy

$$\text{delete}_c(\text{insert}_c(v, m, e), e) = \text{delete}_c(v, m, e) \text{ igaz-e.}$$

Alkalmazzuk az egyenlet bal oldalára a φ leképezést.

$$\varphi(\text{delete}_c(\text{insertc}(v, m, e), e)) = \text{delete}(\varphi(\text{insert}_c(v, m, e)), e) =$$

$$\text{delete}(\text{insert}(\varphi(v, m), e), e) = \text{delete}(\varphi(v, m), e) = \varphi(\text{delete}_c(v, m, e)).$$

Ha tehát feltesszük, hogy azokat a konkrét objektumokat, amelyeknek a φ szerinti képe azonos, nem különböztetjük meg, azaz ha feltesszük, hogy

$$\varphi(c_1) = \varphi(c_2) \Rightarrow c_1 = c_2,$$

akkor

$$\text{delete}_c(\text{insert}_c(v, m, e), e) = \text{delete}_c(v, m, e).$$

Úgy tűnik, általános igazság, hogy

szimuláció teljesülése ≡ szemantika azonossága.

DEFINÍCIÓ Legyen $d_a = (A, F, E_a)$; absztrakt specifikáció, ahol

$$A = \{A_0, A_1, \ldots, A_n\}; \qquad F = \{f_0, f_1, \ldots, f_m\};$$

$$E_a = \{f_s(f_0) = f_0, \ldots, \alpha_j(a) \Rightarrow f_s(f_c(a)) = h_j(\ldots, f_i(a), \ldots; a), \ldots)\}.$$

Legyen adott a $d_c = (C, G, E_c)$ konkrét specifikáció,

$$C = \{C_0, C_1, \ldots, C_n\}; \qquad G = \{g_0, g_1, \ldots, g_m\};$$

ahol C az A egy reprezentációja adott φ: $C \rightarrow A$ mellett.

Ha $E_c = \{g_s(g_0) = g_0, \ldots, \alpha_j'(c) \Rightarrow g_s(g_c(c)) = h_j'(\ldots, g_i(c), \ldots; c), \ldots)\}$,

úgy, hogy

$$(\forall j)((h_j(\ldots, f_i(\varphi(c)), \ldots; \varphi(c)) = \varphi(h_j'(\ldots, g_i(c), \ldots; c)) \wedge$$
$$(\alpha_j(\varphi(c)) = \text{"true"} \equiv \alpha_j'(c)) = \text{"true"})),$$

akkor azt mondjuk, hogy az absztrakt adattípus és a konkrét adattípus szemantikája azonos.

Például:

e_{a_i}: $(e_1 = e_2 \wedge has(s, e_2) = \text{"false"}) \Rightarrow delete(insert(s, e_1), e_2) = s$

e_{c_i}: $(e_1 = e_2 \wedge (\exists i, 1 \leq i \leq m)(access(v, i) = e_2) = \text{"false"}) \Rightarrow$

$$delete_c(insert_c(v, m, e_1), e_2) = (v, m).$$

$s = \varphi(v, m)$;

$(\exists i, 1 \leq i \leq m)(access(v, i) = e_2) \equiv has_c(v, m, e_2)$;

$(e_1 = e_2 \wedge has(\varphi(v, m), e_2) = \text{"false"}) \equiv (e_1 = e_2 \wedge has_c(v, m, e_2)) = \text{"false"}$;

5.5.1. ÁLLÍTÁS (Azonos szemantika esetén teljesül a szimuláció). Legyen megadva a d_a absztrakt és a d_c konkrét specifikáció, valamint a φ: $C \rightarrow A$ reprezentációs függvény, amelyre teljesülnek az

$$f_0 = \varphi(g_0),$$

$$(\forall f_c \in F_c)(f_c(\varphi(c)) = \varphi(g_c(c)))$$

feltételek.

"E_a" a következő formában adott:

$$E_a = \{f_s(f_0) = f_0, \ldots, \alpha(a) \Rightarrow f_s(f_c(a)) = h(\ldots, f_i(a), \ldots; a), \ldots)\},$$

"E_c" hasonlóképpen:

$$E_c = \{g_s(g_0) = g_0, \ldots, \alpha'(c) \Rightarrow g_s(g_c(c)) = h'(\ldots, g_i(c), \ldots; c), \ldots)\},$$

ahol

$$h(\ldots, f_i(\varphi(c)), \ldots; \varphi(c)) = \varphi(h'(\ldots, g_i(c), \ldots; c))$$

és

$$\alpha(\varphi(c)) = \text{"true"} = \alpha'(c)) = \text{"true"}$$

teljesül. Ha mindezek igazak, akkor a d_c konkrét specifikáció a d_a absztrakt specifikáció szerint helyes.

BIZONYÍTÁS Azt kell bizonyítani, hogy

1. "C" az "A" egy reprezentációja az adott $\varphi(c)$ mellett. Ez a konstrukciós műveletekre tett feltevés alapján és az elégséges feltételünk állítása szerint igaz.
2. Valahányszor $f_s(\varphi(c))$ értelmezve van, akkor értelmezve van $g_s(c)$ is. Ez is teljesül. Ugyanis $f_s(\varphi(c))$ értelmezési tartományát

   $$\alpha(\varphi(c)) \wedge \varphi(c) = f_c(\varphi(c'))$$

   határozza meg. Márpedig $\alpha(\varphi(c)) = \text{"true"} \equiv \alpha'(c) = \text{"true"}$, másrészt $\varphi(c) = f_c(\varphi(c')) \equiv c = g_c(c')$, alapján nyilvánvaló, hogy ekkor $g_s(c)$ is értelmezve van.
3. Be kell még látni, hogy $f_s(\varphi(c)) = \varphi(g_s(c))$. Ezt teljes indukcióval látjuk be.

   a.) $c = g_0$.

   Ekkor $f_s(\varphi(c)) = f_s(\varphi(g_0)) = f_s(f_0) = f_0 = \varphi(g_0) = \varphi(g_s(g_0)) = \varphi(g_s(c))$.

b.) $c = g_c(c')$, ahol $f_s(\varphi(c')) = \varphi(g_s(c'))$.

Ezért $f_s(\varphi(c)) = f_s(\varphi(g_c(c'))) = f_s(f_c(\varphi(c'))) = h(\varphi(c')) = \varphi(h'(c')) = \varphi(g_s(g_c(c'))) = \varphi(g_s(c))$.

5.5.2. ÁLLÍTÁS (Szimuláció teljesülése esetén azonos a szemantika).). Legyen megadva a d_a absztrakt és a d_c konkrét specifikáció, valamint a φ: $C \rightarrow A$ reprezentációs függvény, amelyre teljesülnek a

$$f_0 = \varphi(g_0),$$

$$(\forall f_c \in F_c)(f_c(\varphi(c)) = \varphi(g_c(c)))$$

feltételek.

"E_a" a következő formában adott:

$$E_a = \{f_s(f_0) = f_0, \ldots, \alpha(a) \Rightarrow f_s(f_c(a)) = h(\ldots, f_i(a), \ldots; a), \ldots)\},$$

Tegyük fel, hogy

$$(\forall f_s \in F_s)(f_s(\varphi(c)) = \varphi(g_s(c))),$$

a $h(\ldots, f_i(a), \ldots, a)$ kifejezések olyanok, hogy azokat

$h(\ldots, f_i(\varphi(c)), \ldots; \varphi(c))$ formára hozva teljesül a

$$h(\ldots, f_i(\varphi(c)), \ldots; \varphi(c)) = \varphi(h(\ldots, g_i(c), \ldots; c))$$

és $\alpha(\varphi(c))$ esetén pedig az

$$\alpha(\varphi(c)) = \text{"true"} = \alpha'(c)) = \text{"true"},$$

összefüggés.

Tegyük fel továbbá, hogy nem teszünk különbséget két konkrét objektum között, ha a képük azonos:

$$\varphi(c_1) = \varphi(c_2) \Rightarrow c_1 = c_2.$$

Ha mindezek igazak, akkor a d_c konkrét specifikáció jelentése:

$$E_c = \{g_s(g_0) = g_0, \ldots, \alpha'(c) \Rightarrow g_s(g_c(c)) = h(\ldots, g_i(c), \ldots; c), \ldots)\}.$$

BIZONYÍTÁS Legyen

$$c_1 = g_s(g_c(c)), \quad c_2 = h(\ldots, g_i(c), \ldots; c).$$

Bizonyítsuk be, hogy

$$\varphi(c_1) = \varphi(c_2).$$

Az egyenlőség bizonyítása:

$$\varphi(c_2) = \varphi(h(\ldots, g_i(c), \ldots; c)) = h(\ldots, f_i(\varphi(g_i(c))), \ldots; \varphi(c)) = f_s(f_c(\varphi(c))) =$$

$$f_s(\varphi(g_c(c))) = \varphi(g_s(g_c(c))) = \varphi(c_1),$$

azaz

$$g_s(g_c(c)) = h(\ldots, g_i(c), \ldots; c).$$

MEGJEGYZÉS Két konkrét objektum egyenlőségét és azok képének egyenlőségét a következőképpen definiáltuk:

$$\varphi(c_1) = \varphi(c_2) \equiv (\varphi(c_1) = f_0 \wedge \varphi(c_2) = f_0) \vee (\forall f_s \in F_s)(f_s(\varphi(c_1)) = f_s(\varphi(c_2)));$$

$$c_1 = c_2 \equiv (c_1 = g_0 \wedge c_2 = g_0) \vee (\forall g_s \in G_s)(g_s(c_1) = g_s(c_2)).$$

Általában tehát nem igaz, hogy $\varphi(c_1) = \varphi(c_2) \Rightarrow c_1 = c_2$.

5.6. PROCEDURÁLISAN ADOTT KONKRÉT SPECIFIKÁCIÓ ELŐ- ÉS UTÓFELTÉTELEKKEL ADOTT ABSZTRAKT SPECIFIKÁCIÓ SZERINTI HELYESSÉGE

5.6.1. ÁLLÍTÁS Adottak a d_a és a d_c specifikációk közös szignatúrával:

$$d_a = (A, F, E_a); \text{ ahol } f_i \in F;$$

$$\{\text{"true"}\}\ a = f_0\ \{post_{f_0}(a)\},$$

$$\{pre_{f_i}(a)\}\ a' = f_i(a)\ \{post_{f_i}(a, a')\} \in E_a,\ i = 1, \ldots, n;$$

$$d_c = (C, G, E_c);\ Q_{g_i} \in E_c\ ,\ g_i \in G,\ i = 1, \ldots, n;$$

Legyen az absztrakt invariáns

$$A = \{a \mid I_a(a)\},$$

a konkrét invariáns

$$C = \{c \mid I_c(c)\}.$$

Legyenek a konkrét szemantika eljárásai a következők:

procedure g_0 **begin** Q_0 **end**;

procedure g_i **begin** Q_i **end;** i = 1, ..., n;

A reprezentációs függvény:

$$\varphi: C \rightarrow A.$$

Ha a következő tételek teljesülnek:

1. $(\forall c \in C)(I_c(c) \Rightarrow I_a(\varphi(c))$;
2. $\{\text{"true"}\}\ Q_0\ \{post_{f_0}(\varphi(c)) \wedge I_c(c)\}$;
3. $(\forall f \in F)$: $\{pre_{f_i}(\varphi(c)) \wedge I_c(c)\}\ Q_i\ \{post_{f_i}(\varphi(c), \varphi(c')) \wedge I_c(c')\}$;

ahol 2. és 3. teljes helyességi tételek, akkor a d_c konkrét specifikáció helyes a d_a absztrakt specifikáció szerint.

BIZONYÍTÁS Az állítás bizonyítása az elő- és utófeltételes formában adott konkrét specifikáció esetéhez hasonló módon végezhető el.

Az adattípus konkrét specifikációjának az absztrakt specifikáció szerinti helyességét tehát ebben az esetben visszavetettük programok helyességének a bizonyítására. Ehhez tisztáznunk kell a program helyességével kapcsolatos kérdéseket, és módszert kell adnunk a programok tulajdonságainak bizonyítására.

6. SZEKVENCIÁLIS PROGRAMOK HELYESSÉGÉNEK BIZONYÍTÁSA

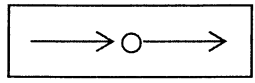
Először a program fogalmának tisztázásával foglalkozunk. A szekvenciális programokat két csoportba szokás osztani:

- determinisztikus programok és
- nem determinisztikus programok.

A determinisztikus programok között megkülönböztethetünk valódi programokat és azokon belül strukturált programokat. A valódi programot a hozzárendelhető gráfja alapján szoktuk definiálni.

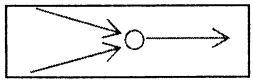
DEFINÍCIÓ A valódi program gráfja a következő tulajdonságokkal rendelkezik:

1. A program gráfja egy egyszeresen összefüggő irányított gráf.
2. A csomópontok és az élek viszonya ötféle lehet:
   - A csomópontba egy él mutat be, és egy él mutat ki, amely a függvénycsomópont esete:

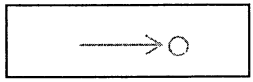


- A csomópontba egy él mutat be, és két él mutat ki, amely a feltételtől függő elágazás esete:

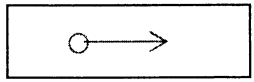
- A csomópontba két él mutat be és egy él mutat ki, amely a gyűjtő csomópont esete:



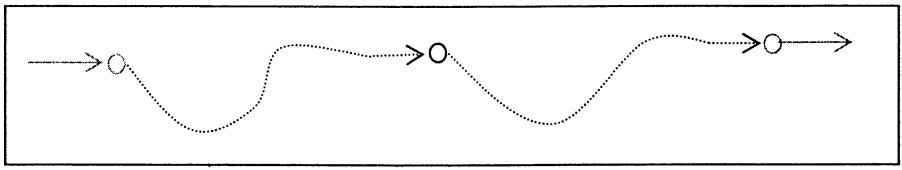
- A gráfban pontosan egy olyan él van, amely egy csomópontba úgy mutat, hogy másik csomópontból nem mutat ki. Ez a csomópont az indítási csomópont:



- A gráfban pontosan egy olyan él van, amely egy csomópontból úgy mutat ki, hogy másik csomópontba nem mutat be. Ez a csomópont a befejezési csomópont:



3. A gráfnak nincs "felesleges" csomópontja, azaz minden csomóponton át legalább egy olyan út vezet a gráfban, amely az indítási csomópontból a befejezési csomópontba visz.



A strukturált program természetesen szintén eleget tesz a valódi program most megfogalmazott követelményeinek, de a strukturált program definíciója tartalmaz további megszorításokat. Ezek alapján lehetővé válik az, hogy a strukturált programok halmazát rekurzív módon definiáljuk.

A STUKTURÁLT PROGRAM SZINTAXISA

S::= **skip** | $y \leftarrow f(x, y)$ | $S_1; S_2$ | **if** $\alpha(x, y)$ **then** S_1 **else** S_2 **fi** |
while $\alpha(x, y)$ **do** S_1 **od**

S, S_1, S_2, determinisztikus programok;

X az S program bemenő adatainak halmaza és $x \in X$;

Y az S program változóinak halmaza és $y \in Y$;

f(x, y): kifejezés;

$\alpha(x, y)$: az x ill. y változóktól függő kvantorfüggetlen logikai kifejezés.

A strukturált program egy kézenfekvő általánosítása a nem determinisztikus program. Ennek a programformának a számítógépek párbeszédes (interaktív) alkalmazásai adták meg a jelentőségét. Ezekben az alkalmazásokban merült fel például a nem determinisztikus kiválasztás lehetőségének a beépítése a programkészítés eszköztárába.

A NEM DETERMINISZTIKUS PROGRAM SZINTAXISA

S::= **skip** | b $\rightarrow$ y $\leftarrow$ f(x, y) | S_1; S_2 | **if** $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **if** |
do $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **od** | **do** $S_B \square S_e$; **exit od**

$\alpha_i(x, y)$: ún. filter vagy őrfeltétel, amely az x ill. y változóktól függő kvantorfüggetlen logikai kifejezés.

if $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **if** nem determinisztikus kiválasztás;

do $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **od** nem determinisztikus iteráció;

do $S_B \square S_e$; **exit od** kilépéses nem determinisztikus iteráció, ahol S_B az iteráció $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ magjának a jelölése.

A következőkben ennek az általános programstruktúrának a jelentését definiáljuk.

A NEM DETERMINISZTIKUS PROGRAM SZEMANTIKÁJA

A szemantikát állapottranszformációs módszerrel definiáljuk.

Jelölések: Az S program állapotainak halmazát jelölje: Σ.

Egy állapot jele: $\sigma \in \Sigma$. Az állapotátmenetet a következőképpen jelöljük:

$$<S, \sigma> \rightarrow <S', \sigma'>,$$

ami azt jelenti, hogy az S program a σ kiindulási állapotból az S' maradék programba és a σ' eredményállapotba jut el.

Ha a program a σ' állapotban befejeződik, akkor S' helyére T-t írunk.

Üres program jele a program végrehajtásán belül E.

Legyen $\sigma \equiv$ fail a végrehajtás közbeni hibát reprezentáló állapot, $\sigma \equiv \perp$ pedig virtuális állapot, amely a divergenciát reprezentálja.

AXIÓMÁK A determinisztikus, strukturált programszerkezetek szemantikáját az állapotátmenetekhez tartozó axiómákkal definiálhatjuk.

$$\langle \textbf{skip}, \sigma \rangle \rightarrow \langle E, \sigma \rangle.$$

$$\langle y \leftarrow f(x, y), \sigma \rangle \rightarrow \langle E, \sigma(y \leftarrow f(x, y)) \rangle,$$

itt tehát $\sigma' = \sigma(y \leftarrow f(x, y))$ azt az állapotot jelöli, amelyet a σ állapotból kapunk az $y \leftarrow f(x, y)$ értékadás eredményeként.

$$\frac{\langle S_1, \sigma_1 \rangle \rightarrow \langle S_2, \sigma_2 \rangle}{\langle S_1; S, \sigma_1 \rangle \rightarrow \langle S_2; S, \sigma_2 \rangle}$$

Ez szóban azt jelenti, hogy ha $\langle S_1, \sigma_1 \rangle \rightarrow \langle S_2, \sigma_2 \rangle$ teljesül, akkor abból következik, hogy $\langle S_1; S, \sigma_1 \rangle \rightarrow \langle S_2; S, \sigma_2 \rangle$ is teljesülni fog.

$\sigma(\alpha) = \text{"true"} \Rightarrow \langle \textbf{if } \alpha \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle,$

$\sim\sigma(\alpha) = \text{"true"} \Rightarrow \langle \textbf{if } \alpha \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle.$

$\sigma(\alpha(x, y)) = \text{"true"}$ azt az állapotot jelenti, amelyre $\alpha = \text{"true"}$.

$\sigma(\alpha) = \text{"true"} \Rightarrow \langle \textbf{while } \alpha \textbf{ do } S \textbf{ od}, \sigma \rangle \rightarrow$

$$\langle S; \textbf{while } \alpha \textbf{ do } S \textbf{ od}, \sigma \rangle$$

$\sim\sigma(\alpha) = \text{"true"} \Rightarrow \langle \textbf{while } \alpha \textbf{ do } S \textbf{ od}, \sigma \rangle \rightarrow \langle E, \sigma \rangle$

Kézenfekvő megállapítás az, hogy bármely S determinisztikus program esetén egy adott σ állapotból kiindulva pontosan egy kiszámítási szabály létezik.

AXIÓMÁK A nem determinisztikus programszerkezetekhez tartozó állapotátmenetek.

$$\sigma(b) = \text{"true"} \Rightarrow \langle b \rightarrow y \leftarrow f(x, y), \sigma\rangle \rightarrow \langle E, \sigma(y \leftarrow f(x, y))\rangle$$

$$\sim\sigma(b) = \text{"true"} \Rightarrow \langle b \rightarrow y \leftarrow f(x, y), \sigma\rangle \rightarrow \langle E, fail\rangle,$$

$$(\exists i \in \{1,\ldots,n\})(\sigma(\alpha_i) = \text{"true"}) \Rightarrow$$

$$\langle \textbf{if } \alpha_1 \rightarrow S_1 \,\square\, \ldots \,\square\, \alpha_n \rightarrow S_n \textbf{ fi}, \sigma\rangle \rightarrow \langle S_i, \sigma\rangle,$$

$$(\sim\exists i \in \{1, \ldots, n\})(\sigma(\alpha_i) = \text{"true"}) \Rightarrow$$

$$\langle \textbf{if } \alpha_1 \rightarrow S_1 \,\square\, \ldots \,\square\, \alpha_n \rightarrow S_n \textbf{ fi}, \sigma\rangle \rightarrow \langle E, fail\rangle,$$

$$(\exists i \in \{1, \ldots, n\})(\sigma(\alpha_i) = \text{"true"}) \Rightarrow$$

$$\langle \textbf{do } \alpha_1 \rightarrow S_1 \,\square\, \ldots \,\square\, \alpha_n \rightarrow S_n \textbf{ od}, \sigma\rangle \rightarrow$$

$$\langle S_i; \textbf{ do } \alpha_1 \rightarrow S_1 \,\square\, \ldots \,\square\, \alpha_n \rightarrow S_n \textbf{ od}, \sigma\rangle,$$

$$(\sim\exists i \in \{1, \ldots, n\})(\sigma(\alpha_i) = \text{"true"}) \Rightarrow$$

$$\langle \textbf{do } \alpha_1 \rightarrow S_1 \,\square\, \ldots \,\square\, \alpha_n \rightarrow S_n \textbf{ od}, \sigma\rangle \rightarrow \langle E, \sigma\rangle,$$

Az S_0 program egy végrehajtási sorozata a σ_0 kiindulási állapottal a következő:

τ: $\langle S_0, \sigma_0\rangle \rightarrow \langle S_1, \sigma_1\rangle \rightarrow \ldots$

- minden $\langle S_i, \sigma_i\rangle \rightarrow \langle S_{i+1}, \sigma_{i+1}\rangle$ átmenethez tartozik egy $(S_i, \alpha_i \rightarrow r_i, S_{i+1})$ tranzakció, úgy hogy $\alpha(\sigma_i) = \text{"true"}$ és $\sigma_{i+1} = r_i(\sigma_i)$;
- a τ végrehajtás lehet véges, ha az utolsó eleme a sorozatnak $\langle S_n, \sigma_n\rangle$ és $\sigma_n \in \Sigma$. Jelölés: $Val(\tau) = \sigma_n$;
- a τ végrehajtás lehet véges, de az utolsó eleme a sorozatnak $\langle S_n, \sigma\rangle$ és $S_n \neq T$. Jelölés: $Val(\tau) = fail$. $fail \notin \Sigma$.
- a τ végrehajtás lehet végtelen (divergens). Jelölés: $Val(\tau) = \perp$. $\perp \notin \Sigma$.

Jelölje comp(S)(σ) az S végrehajtás összes kiszámítását, amely a σ kezdési állapothoz tartozik.

AZ S PROGRAM JELENTÉSE:

$$M[S](\sigma) = \{val(\tau) \mid \tau \in comp(S)(\sigma)\},$$

Ha az S végrehajtása **sikertelen**: $fail \in M[S](\sigma)$.

Ha az S végrehajtása **divergens**: $\perp \in M[S](\sigma)$.

Jelölje P(Σ) az S program helyes eredményállapotainak a halmazát parciális helyesség esetén. Akkor az S program **parciális helyességi szemantikája** a következő leképezés:

$$M[S]: \Sigma \rightarrow P(\Sigma).$$

Az S program **teljes helyességi szemantikája** pedig a következő:

$$M[S]_{tot}: \Sigma \rightarrow (P(\Sigma) \cup \{\perp\}).$$

6.1. A PROGRAMHELYESSÉG FOGALMAI

Legyen **a P program specifikációja** egy (φ, ψ) kettős, ahol φ a program előfeltétele és ψ a program utófeltétele, azaz $\varphi(\sigma_0)$ = "true" és $\forall \sigma' \in M[P](\sigma_0)$ esetén $\psi(\sigma')$ = "true".

PARCIÁLIS HELYESSÉG Az S programot a (φ, ψ) specifikáció szerint parciálisan helyesnek mondjuk, ha minden olyan $\sigma \in \Sigma$ kezdőértékre, amelyre $\varphi(\sigma)$ = "true" feltéve, hogy ha a végrehajtás befejeződik, akkor $\sigma' \in \Sigma$ és $\psi(\sigma')$ = "true", azaz:

$$[(\varphi(\sigma) = \text{"true"}) \wedge (\sigma' \in M[S](\sigma))] \Rightarrow \psi(\sigma') = \text{"true"}.$$

Jelölése: {φ} P {ψ}.

EREDMÉNYESSÉG $\varphi(\sigma) \Rightarrow (fail \notin M[S](\sigma))$.

BEFEJEZŐDÉS $\varphi(\sigma) \Rightarrow (\perp \notin M[S](\sigma))$.

TELJES HELYESSÉG A S programot a (φ, ψ) specifikáció szerint teljes helyesnek mondjuk, ha minden olyan $\sigma \in \Sigma$ kezdőértékre, amelyre $\varphi(\sigma)$ = "true", a tranzakció befejeződik és a $\sigma' \in \Sigma$, amelyre $\psi(\sigma')$ = "true":

- $\varphi(\sigma) \Rightarrow (\{\perp, \text{fail}\} \cap M[S](\sigma) = \{\})$;
- $[(\varphi(\sigma) = \text{"true"}) \wedge (\sigma' \in M[S](\sigma))] \Rightarrow \psi(\sigma') = \text{"true"}$.

Jelölése: $\{\{\varphi\}\}\ P\ \{\{\psi\}\}$.

6.2. A PROGRAM HELYESSÉGI TULAJDONSÁGAINAK BIZONYÍTÁSA

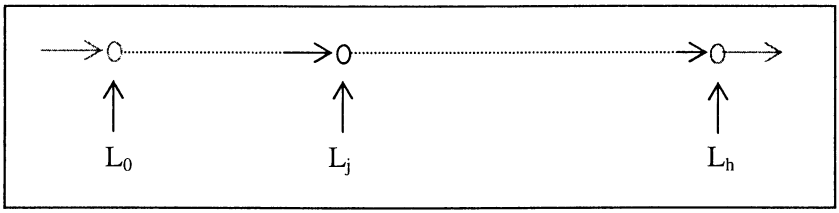
6.2.1. VALÓDI PROGRAMOK PARCIÁLIS HELYESSÉGÉNEK BIZONYÍTÁSA

Először foglalkozzunk a valódi program parciális helyességének bizonyításával. Adott a

$$\{P(x)\}\ S\ \{R(x, x')\}$$

parciális helyességi tétel. Ennek a tételnek a bizonyítására Floyd [FLO67] a következő módszert javasolta.

1. Helyezzünk el a programban az indítási ponthoz, a befejezési ponthoz és a programban különböző pontokhoz címkéket úgy, hogy minden ciklust szakítson meg legalább egy címke.



2. Minden L_i címkéhez helyezzük el az abban a pontban az x bemenő adatok alapján keletkező x' részeredmény specifikációját:

$$Q_0(x, x) \qquad Q_j(x, x') \qquad Q_h(x, x')$$

3. Minden L_i címkétől az L_j címkéig terjedő olyan α_{ij} útszakaszra, amelyen nem található újabb címke, a program szövege alapján határozzuk meg
   - annak a feltételét, hogy ez az útszakasz végrehajtódik: $R_{\alpha_{ij}}(x, x')$;
   - a végrehajtás eredményeként megvalósuló függvényt: $r_{\alpha_{ij}}(x, x')$.
4. Ha bebizonyítjuk, hogy

$$P(x) \Rightarrow Q_0(x, x),$$
$$(\forall \alpha_{ij})(Qi(x, x') \wedge R_{\alpha_{ij}}(x, x') \Rightarrow Q_j(x, r_{\alpha_{ij}}(x, x'))$$
$$Q_h(x, x') \Rightarrow R(x, x')$$

akkor az S program az adott specifikáció szerint parciálisan helyes.

BIZONYÍTÁS Az állítás következik a valódi program definíciójából és abból, hogy a program végrehajtásai a program gráfján a belépési ponttól a kilépési ponthoz vezető utak mentén valósulnak meg. Minden ilyen út mentén bebizonyítjuk, hogy ha ezek az utak végrehajtódnak, akkor ezek mentén minden részeredmény kielégíti az általunk megfogalmazott állítást, feltéve, hogy a bemenő adat az általunk megadott állításnak felelt meg. A részeredmények között mind-

egyik út végén ott van a végeredményre adott állításunk is. Ezzel a parciális helyességi tétel állítását bebizonyítottuk.

6.2.2. VALÓDI PROGRAMOK TELJES HELYESSÉGÉNEK BIZONYÍTÁSA

Definíció. Jól rendezett halmaz. Legyen W egy halmaz és $\langle$: W×W bináris reláció. A $\langle$ relációt rendezettnek mondjuk, ha a, b, c ∈ W- re teljesülnek a következők:

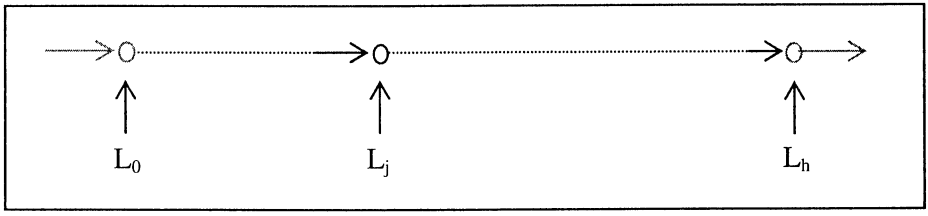
- Irreflexív: a $\langle$ a = "false".
- Antiszimmetrikus: a $\langle$ b = "true" $\Rightarrow$ b $\langle$ a = "false".
- Tranzitív: (a $\langle$ b = "true") $\wedge$ (b $\langle$ c = "true") $\Rightarrow$ a $\langle$ c = "true".

A $W_{\langle}$ halmazt jól rendezettnek nevezzük, ha nem létezik végtelen ... $\langle w_2 \langle w_1 \langle w_0$, sorozat $w_i \in W_{\langle}$, i = 0, 1, 2, ...esetén.

Adott a teljes helyességi tétel:

$$\{\{P(x)\}\}\ S\ \{\{R(x, x')\}\}$$

Helyezzünk el a programban az indítási ponthoz, a befejezési ponthoz és a programban különböző pontokhoz címkéket úgy, hogy minden ciklust szakítson meg legalább egy címke.



1. Minden L_i címkéhez helyezzük el az abban a pontban az x bemenő adatok alapján keletkező x' részeredmény specifikációját:

$Q_0(x, x)$ $Q_j(x, x')$ $Q_h(x, x')$

2. Válasszunk egy alkalmas reláció mellett jól rendezett halmazt. Legyen ez a $W_<$ a $<$ reláció mellett.
3. Minden L_i címkéhez válasszunk meg egy $E_i(x, x')$ függvényt, amelyre igaz, hogy

$$Ei(x, x') \in W_<$$

4. Minden L_i címkétől az L_j címkéig terjedő olyan α_{ij} útszakaszra, amelyen nem található újabb címke, a program szövege alapján határozzuk meg:
   - annak a feltételét, hogy ez az útszakasz végrehajtódik: $R_{\alpha_{ij}}(x, x')$;
   - a végrehajtás eredményeként megvalósuló függvényt: $r_{\alpha_{ij}}(x, x')$.
5. Ha bebizonyítjuk, hogy

$$P(x) \Rightarrow Q_0(x, x);$$

$$(\forall i)(Q_i(x, x') \Rightarrow E_i(x, x') \in W_<;$$

$$(\forall \alpha_{ij})(Q_i(x, x') \wedge R_{\alpha_{ij}}(x, x') \Rightarrow$$

$$(Q_j(x, r_{\alpha_{ij}}(x, x')) \wedge E_i(x, x') < Ei(x, r_{\alpha_{ij}}(x, x'))));$$

$$Q_h(x, x') \Rightarrow R(x, x');$$

akkor az S program az adott specifikáció szerint teljesen helyes.

BIZONYÍTÁS Az állítás következik a parciális helyesség bizonyításánál mondottakból és abból a tényből, hogy $W_<$ jól rendezett halmaz, azaz a $<$ reláció mellett nem létezik végtelen hosszú sorozat.

MEGJEGYZÉSEK:

- Elég csak az iterációkban elhelyezett címkékhez $E_i(x, x')$ függvényeket választani.

- Az iteráció esetén általában a kisebb reláció mellett jó választás lehet

$$W_< = \{j \mid 0 \leq j \leq m\},$$

ahol "m" az "n" ciklusszámláló legnagyobb értéke és a ciklus invariánsa I(x, x', n). Itt x a paramétereket, x' a változókat jelöli.

- Legyen az iterációra vonatkozó tétel a következő:

$$\{\{pre(x, x')\}\}\ \textbf{while}\ \alpha(x, x')\ \textbf{do}\ S\ \textbf{od}\ \{\{post(x, x')\}\}$$

Ekkor az iteráció teljes helyességére vonatkozó tételek a következő egyszerű formát öltik:

$$pre\ (x, x') \Rightarrow I(x, x', 0);$$

$$\{\{I(x, x', n) \wedge \alpha(x, x', n)\}\}\ S\ \{\{I(x, x', n + 1)\}\};$$

$$I(x, x', n) \Rightarrow n \leq m;$$

$$I(x, x', n) \wedge \sim\alpha(x, x', n) \Rightarrow post(x, x');$$

- Ha a cikluson "n" db út vezet keresztül, akkor az E(x, x') függvény például a következő módon választható meg:

Legyen $x' = X \cup \{y_1, \ldots, y_n\}$, ahol, y_i, $i = 1,\ldots, n$; a számunkra alkalmas egész értékű változókat jelöli, X pedig a többi változót. Legyen

$$E(y_1,\ldots, y_n) = \beta_0 + \beta_1 \times y_1 + \ldots + \beta_n \times y_n.$$

Jelölje $R_i(x, x')$ annak a feltételét, hogy a ciklusmagban az i-edik út hajtódik végre, amelynek végrehajtása esetén a programfüggvény $f_{S_i}(x, x')$.

Ekkor a ciklus teljes helyességére vonatkozó tételek a következők:

- $pre(x, x') \Rightarrow I(x, x')$;
- $I(x, x') \wedge \sim\alpha(x, x') \Rightarrow post(x, x')$;
- $I(x, x') \Rightarrow \beta_0 + \beta_1 \times y_1 + \ldots + \beta_n \times y_n \geq 0$;
- $I(x, x') \wedge \alpha(x, x') \wedge R_i(x, x') \wedge z = f_{S_i}(x, x') \Rightarrow I(x, z) \wedge$
  $(\beta_0 + \beta_1 \times y_1 + \ldots + \beta_n \times y_n > \alpha_0 + \alpha_1 \times y_1' + \ldots + \alpha_n \times y_n')$;
  $j = 1,\ldots, n$;

Így tehát $n + 1$ darab egyenlőtlenséget kaptunk az $n + 1$ darab $\beta_0, \beta_1, \ldots, \beta_n$, ismeretlen meghatározására, ami általában egyszerű feladat. A következőkben erre mutatunk példát.

6.2.2.1. PÉLDA Bizonyítsuk be, hogy egész aritmetika esetén igaz a következő tétel:

$$\{x_1 > 0 \wedge x_2 > 0\}$$

$$(y_1, y_2) \leftarrow (x_1, x_2);$$

while $y_1 \neq y_2$ **do**

if $y_1 > y_2$ **then** $y_1 \leftarrow y_1 - y_2$ **else** $y_2 \leftarrow y_2 - y_1$ **fi**;

od;

$$\{y_1 = lnko(x_1, x_2)\}$$

BIZONYÍTÁS: R_0: "true"; $r_0(x, y) = (x_1, x_2)$;

R_{11}: $y_1 > y_2$; $r_{11}(x, y) = (y_1 - y_2, y_2)$;

R_{12}: $y_2 > y_1$; $r_{12}(x, y) = (y_1, y_2 - y_1)$;

R_2: $y_1 = y_2$; $r_2(x, y) = (y_1, y_2)$;

$$Q(x_1, x_2; y_1, y_2): y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge$$
$$x_1 > 0 \wedge x_2 > 0;$$

$$E(x_1, x_2; y_1, y_2) = \beta_0 + \beta_1 \times y_1 + \beta_2 \times y_2;$$

$$x_1 > 0 \wedge x_2 > 0 \wedge y = r_0(x, y) \Rightarrow Q(x_1, x_2; y_1, y_2);$$

$$(x_1 > 0 \wedge x_2 > 0 \wedge y_1 = x_1 \wedge y_2 = x_2) \Rightarrow$$
$$(y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge x_1 > 0 \wedge x_2 > 0);$$

$$Q(x_1, x_2; y_1, y_2) \Rightarrow E(x_1, x_2; y_1, y_2) \in \{0, 1, 2, \ldots\};$$
$$(y_1 > 0 \wedge y_2 > 0 \wedge$$
$$lnko(y_1, y_2) = lnko(x_1, x_2)) \Rightarrow \beta_0 + \beta_1 \times y_1 + \beta_2 \times y_2 \geq 0;$$

$$(y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge y_1 > y_2) \Rightarrow$$
$$(y_1 - y_2 > 0 \wedge y_2 > 0 \wedge lnko(y_1 - y_2, y_2) = lnko(x_1, x_2) \wedge$$
$$(\beta_0 + \beta_1 \times y_1 + \beta_2 \times y_2 > \beta_0 + \beta_1 \times (y_1 - y_2) + \beta_2 \times y_2);$$

$$(y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge y_2 > y_1) \Rightarrow$$
$$(y_1 > 0 \wedge y_2 - y_1 > 0 \wedge lnko(y_1, y_2 - y_1) = lnko(x_1, x_2) \wedge$$
$$(\beta_0 + \beta_1 \times y_1 + \beta_2 \times y_2 > \beta_0 + \beta_1 \times y_1 + \beta_2 \times (y_2 - y_1));$$

$$(y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge y_1 = y_2) \Rightarrow$$
$$y_1 = lnko(y_1, y_2);$$

Az egyenlőtlenségrendszer egy lehetséges megoldása:

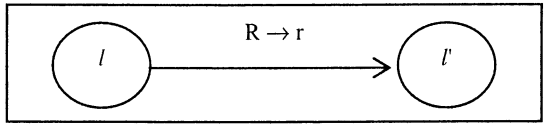
$$\beta_0 = 0; \beta_1 = 1; \beta_2 = 1;$$
$$E(x_1, x_2; y_1, y_2) = y_1 + y_2;$$

6.3. FLOYD MÓDSZERÉNEK FORMÁLIS DEFINÍCIÓJA

A program végrehajtása során a programban elhelyezett címkék között állapotátmenetek játszódnak le. Floyd módszerének az a lényege, hogy a program szövege alapján meghatározzuk ezeknek az átmeneteknek a feltételeit és az állapotok leképezéseit, majd ellenőrizzük a címkéknél elhelyezett invariánsok teljesülését. Most kiterjesztjük ezt a módszert nem determinisztikus programszerkezetekre is.

DEFINÍCIÓ A tranzakciós diagram egy olyan (L, T, s, t) négyes, ahol L a címek egy véges halmaza, $l \in L$; T az $(l, R \rightarrow r, l')$ hármasok egy véges halmaza, amelyeket tranzakcióknak nevezünk.

A tranzakciót szemlélteti a következő ábra:



$l, l' \in L$.

$R: \Sigma \rightarrow bool$.

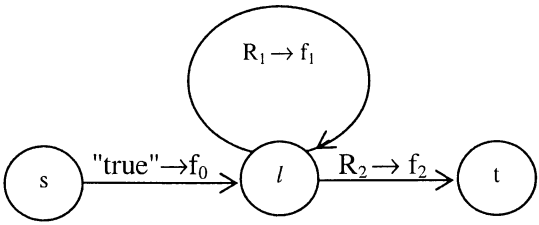
$r: \Sigma \rightarrow \Sigma$, ahol Σ az állapotok halmaza, egy állapot $\sigma \in \Sigma$.

$s \in L$, egy kitüntetett cím, a program belépési pontjának címe.

$t \in L$, egy kitüntetett cím, a program kilépési pontjának címe, amelyre

$$\sim\exists(l \in L, R, t)((t, R \rightarrow r, l) \in T).$$

Példa: P = (L, T, s, t); L ={s, *l*, t};
T = {(s, "true" → f_0, *l*), (*l*, $R_1 \rightarrow f_1$, *l*), (*l*, $R_2 \rightarrow f_2$, t)};



Jelölések: Legyen

$$\text{var} = \{x, y, z, \ldots\},$$

a program változóinak jelölése, és jelölje $\sigma(y)$ az y változó értékét a σ állapotban.

Az

$$(x_1, x_2, x_3, \ldots) \leftarrow (g_1, g_2, g_3, \ldots);$$

állapotátmenet értékadást jelent, a következő komponensekkel:

$$g_i: \Sigma \rightarrow \text{VAR}, i = 1, 2, \ldots$$

A TRANZAKCIÓS DIAGRAM JELENTÉSE A P tranzakciós diagram egy végrehajtási sorozata az l_0 címkétől a σ_0 indulási állapottal a következő:

$$\tau: \langle l_0, \sigma_0\rangle \rightarrow \langle l_1, \sigma_1\rangle \rightarrow \ldots$$

úgy, hogy l_0 = s, továbbá

- minden $\langle l_i, \sigma_i\rangle \rightarrow \langle l_i + 1, \sigma_i + 1\rangle$ átmenethez tartozik egy $(l_i, \alpha \rightarrow r_i, l_i + 1)$ tranzakció úgy, hogy $\alpha(\sigma_i)$ = "true" és $\sigma_{i+1} = r_i(\sigma_i)$;
- a τ végrehajtás lehet véges, ha az utolsó eleme a sorozatnak $\langle l_n, \sigma_n\rangle$ és $\sigma_n \in \Sigma$. Jelölés: $\text{Val}(\tau) = \sigma_n$;
- a τ végrehajtás lehet véges, de az utolsó eleme a sorozatnak $\langle l, \sigma\rangle$ és $l \neq$ t. Jelölés: $\text{Val}(\tau)$ = fail. fail$\notin \Sigma$.

- a τ végrehajtás lehet végtelen (divergens). Jelölés: $Val(\tau) = \perp$, $\perp \notin \Sigma$.

Jelölje comp(P)(σ) a P tranzakciós diagram összes olyan kiszámítását, amely a σ kezdési állapothoz tartozik. A P tranzakciós diagram jelentése:

$$M(P)(\sigma) = \{val(\tau) \mid \tau \in comp(P)(\sigma)\}.$$

Ha a P tranzakciós diagram sikertelen: $fail \in M(P)(\sigma)$.

Ha a P tranzakciós diagram divergens: $\perp \in M(P)(\sigma)$.

A P program specfikációja egy (φ, ψ) kettős, ahol φ a program előfeltétele és ψ a program utófeltétele, azaz $\varphi(\sigma_0)$ = "true" és $\forall \sigma \in M(P)(\sigma_0)$ esetén $\psi(\sigma)$ = "true".

Két P = (L, T, s, t), P' = (L', T', s', t') tranzakciós diagramot akkor nevezünk ekvivalenseknek, ha létezik egy olyan ϕ: L $\rightarrow$ L' bijekció, hogy

- s' = ϕ(s);
- t' = ϕ(t);
- (n, R $\rightarrow$ r, m)$\in$T akkor és csak akkor, ha (ϕ(n), R $\rightarrow$ r, ϕ(m))$\in$T '.

(Például az elnevezés nyilvánvalóan ekvivalens reláció.)

DEFINÍCIÓ A tranzakciós rendszer ekvivalens tranzakciós diagramok egy osztálya. Ezt az ekvivalens osztályt nevezzük **programnak**.

PARCIÁLIS HELYESSÉG A P programot a (φ, ψ) specifikáció szerint parciálisan helyesnek mondjuk, ha minden olyan $\sigma \in \Sigma$ kezdőértékre, amelyre $\varphi(\sigma)$ = "true" feltéve, hogy a tranzakció befejeződik, akkor $\sigma' \in \Sigma$ és $\psi(\sigma')$ = "true":

$$(\varphi(\sigma) = \text{"true"}) \wedge (\sigma' \in M(P)(\sigma) \Rightarrow \psi(\sigma') = \text{"true"}).$$

Jelölése: $\{\varphi\}$ P $\{\psi\}$.

EREDMÉNYESSÉG $\varphi(\sigma) \Rightarrow (\text{fail} \notin M(P)(\sigma))$;

BEFEJEZŐDÉS $\varphi(\sigma) \Rightarrow (\perp \notin M(P)(\sigma))$;

TELJES HELYESSÉG A P programot a (φ, ψ) specifikáció szerint teljes helyesnek mondjuk, ha minden olyan $\sigma \in \Sigma$ kezdőértékre, amelyre $\varphi(\sigma)$ = "true", a tranzakció befejeződik és $\sigma' \in \Sigma$, amelyre $\psi(\sigma')$ = "true":

- $\varphi(\sigma) \Rightarrow (\{\perp, \text{fail}\} \cap M(P)(\sigma) = \varnothing)$;
- $[(\varphi(\sigma) = \text{"true"}) \wedge (\sigma' \in M(P)(\sigma)] \Rightarrow \psi(\sigma') = \text{"true"}$.

Jelölése: $\{\{\varphi\}\}\ P\ \{\{\psi\}\}$.

6.3.1. INDUKTÍV ÁLLÍTÁSOK MÓDSZERE PROGRAMOK PARCIÁLIS HELYESSÉGÉNEK BIZONYÍTÁSÁRA

DEFINÍCIÓ Legyen P = (L, T, s, t) program.

- Egy állításokkal kiegészített diagram a P program számára egy Q függvény, amely minden $l \in L$ címkéhez hozzárendel egy Q_l állítást. Ezt **Q-diagramnak** nevezzük.
- Legyen adva a Q diagram a P programhoz. Egy $\rho = (l, R \rightarrow r, l')$ tranzakció **verifikácós feltétele**:

$$V_\rho = Q_l \wedge R \Rightarrow Q_{l'} \circ r.$$

A P programhoz tartozó Q diagram összes verifikációs feltételének a halmazát jelölje V(P, Q).

- A P programhoz tartozó Q diagramról azt mondjuk, hogy **induktív**, ha minden $V_\rho \in V(P, Q)$ helyes.
- A P programhoz rendelt Q diagramról azt mondjuk, hogy az **invariáns**, ha minden $Q_s(\sigma_0)$ = "true" esetén $Q_{l_i}(\sigma_i)$ = "true".

- Az induktív Q diagramot az adott (φ, ψ) specifikáció szerint **konzisztens**-nek mondjuk, ha $\varphi \Rightarrow Q_s(\sigma_0)$ és $Q_t \Rightarrow \psi$.

6.3.2. A FLOYD-FÉLE INDUKTÍV ÁLLÍTÁSOK MÓDSZERE PROGRAMOK PARCIÁLIS HELYESSÉGÉNEK BIZONYÍTÁSÁRA

1. A P programhoz készítsük el a Q állításokkal kiegészített gráfot.
2. Bizonyítsuk be, hogy az állítások diagramja induktív.
3. Bizonyítsuk be, hogy a Q gráf konzisztens a (φ, ψ) specifikáció szerint.

6.3.3. A FLOYD-FÉLE INDUKTÍV ÁLLÍTÁSOK MÓDSZERE PROGRAMOK TELJES HELYESSÉGÉNEK BIZONYÍTÁSÁRA

Terjesszük ki a módszert programok teljes helyességének bizonyítására is.

1. A P programhoz készítsük el a Q állításokkal kiegészített gráfot.
2. Bizonyítsuk be, hogy az állítások diagramja induktív.

Válasszunk egy jól rendezett halmazt,

- Bizonyítsuk be, hogy

$$(\forall l \in L)\ (Q_l(\sigma) = \text{"true"} \Rightarrow \rho_l(\sigma) \in W_{\langle}) .$$

- Bizonyítsuk be, hogy

$$(\forall (l, R \rightarrow r, l') \in T)(Q_l(\sigma) \wedge \alpha(\sigma) = \text{"true"} \Rightarrow \rho_{l'}(r(\sigma)) \langle \rho_l(\sigma)).$$

- Bizonyítsuk be, hogy a Q gráf konzisztens a (φ, ψ) specifikáció szerint.

6.3.3.1. Példa. Adva van a nat típusosztály specifikációja:

nata **is a class specification** =

parameters = **bool** +

exports =

class sorts: nata

oprs: zerus: $\rightarrow$ nata

succ: nata $\rightarrow$ nata

prec: nata → nata

add: nata nata → nata

eqns: n, m ∈ nata;

add(n, m) = add(m, n)

add(n, zerus) = n

prec(zerus) = " undefined"

prec(succ(n)) = n

add(n, succ(m)) = succ(add(n, m)).

imports =

sorts: natc

oprs: 0: → natc

_ + 1: natc → natc

_ + _: natc natc → natc

=: natc natc → bool[infix]

eqns: i, j, k∈natc;

j + 0 = j

i + j = j + i

i + (j + 1) = (i + j) + 1

i = j = j = i

$i = j \wedge j = k \Rightarrow i = k$

body =

oprs: rep: natc → nata

eqns: a∈natc;

zerus = 0

succ(a) = a + 1

end nata;

Adjuk meg a reprezentációs függvény rekurzív definícióját:

$$\varphi(a) = \textbf{if } a = 0 \textbf{ then } \text{zerus } \textbf{else } \text{succ}(\varphi(a - 1)) \textbf{ fi};$$

Tegyük fel, hogy a törzsrészben például az a + b összeadást a következő algoritmussal kívánjuk megvalósítani:

Q^+:

begin a, b, c, s$\in \text{nat}_c$;

(c, s) $\leftarrow$ (a, 0);

while s $\neq$ b **do**

(c, s) $\leftarrow$ (c + 1, s + 1);

od;

end;

Adjuk meg ennek az algoritmusnak a helyességére vonatkozó tételt, és bizonyítsuk be annak teljes helyességét a Floyd-módszerrel.

MEGOLDÁS Axiómák:

add(n, m) = add(m, n)

add(n, zerus) = n

add(n, succ(m)) = succ(add(n, m)).

Az axiómák átalakítása elő- és utófeltételes formára:

$$\{k = \text{zerus} \vee k = \text{succ}(m)\}$$

$$y = \text{add}(n, k)$$

$$\{(k = \text{zerus} \wedge y = n) \vee (k = \text{succ}(m) \wedge y = \text{succ}(\text{add}(n, m)))\}.$$

$$\{k = \text{zerus} \vee k = \text{succ}(n)\}$$

$$y = \text{add}(k, m)$$

$$\{(k = \text{zerus} \wedge y = m) \vee (k = \text{succ}(n) \wedge y = \text{succ}(\text{add}(n, m)))\}.$$

Bizonyítandó **tétel** az átírt

$$\{k = zerus \vee k = succ(n)\}$$

$$y = add(k, m)$$

$$\{(k = zerus \wedge y = m) \vee (k = succ(n) \wedge y = succ(add(n, m)))\}$$

axióma alapján a következő:

$$\{(\varphi(a) = zerus \vee \varphi(a) = succ(n)) \wedge a \in nat_c \wedge b \in nat_c\}$$

$$(c, s) \leftarrow (a, 0);$$

while $s \neq b$ **do**

$$(c, s) \leftarrow (c + 1, s + 1);$$

od;

$$\{(\varphi(a) = zerus \wedge \varphi(c) = \varphi(b)) \vee$$

$$(\varphi(a) = succ(n) \wedge \varphi(c) = succ(add(n, \varphi(b)))) \wedge c \in nat_c\}.$$

Ennek a teljes helyességi bizonyítására ezután alkalmazhatjuk a Floyd-módszert.

A TÉTEL BIZONYÍTÁSA A program most

$$P = (L, T, s, t);$$

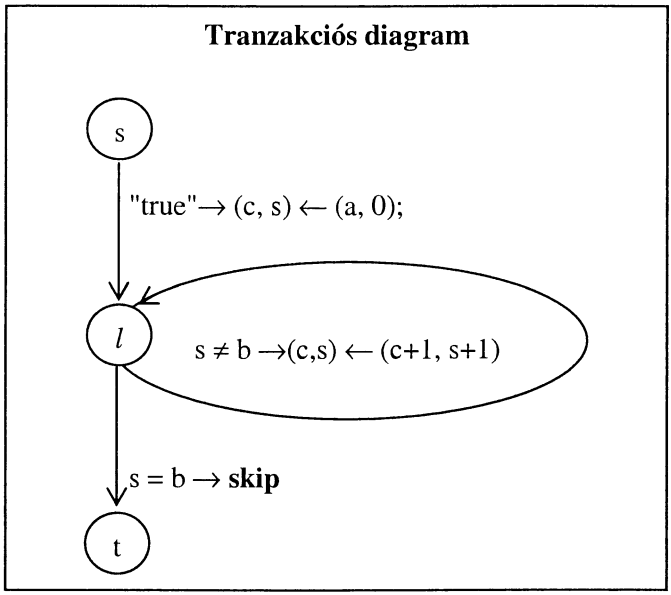
ahol

$$L = \{s, l, t\};$$

$$T = \{(s, \text{"true"} \rightarrow (c, s) \leftarrow (a, 0), l),$$

$$(l, s \neq b \rightarrow (c, s) \leftarrow (c + 1, s + 1), l), (l, s = b \rightarrow \textbf{skip}, t)\}.$$

Ennek alapján készítsük el először a tranzakciós diagramot.

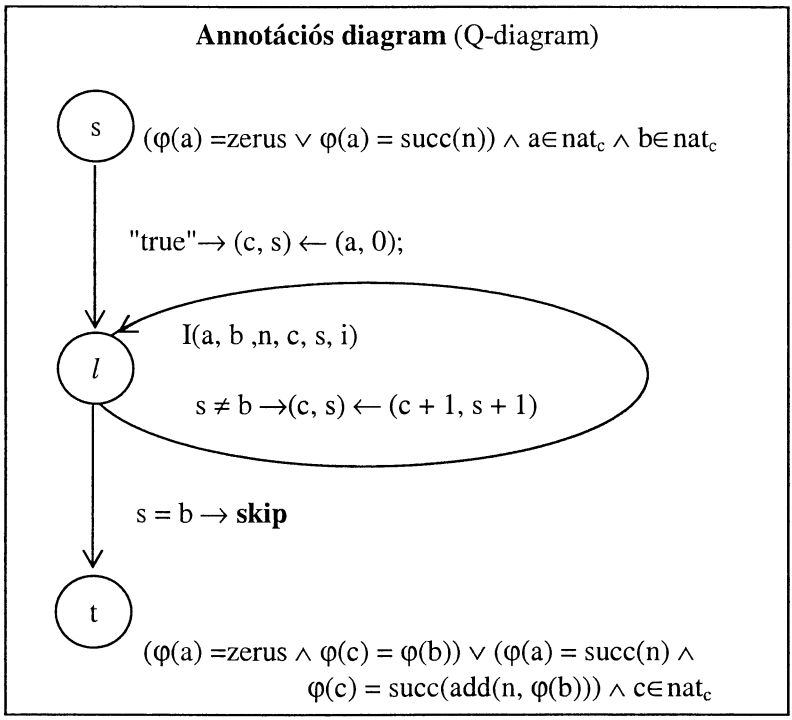


Határozzuk meg a programhoz a ciklus invariánsát. A jól rendezett halmaz most a nem negatív egész számok halmaza lehet a < reláció mellett, amelyen a ciklusszámláló van értelmezve.

$$I(a, b, n, c, s, i): ((\varphi(a) = zerus) \vee \varphi(a) = succ(\varphi(n))) \wedge a \in nat_c \wedge b \in nat_c) \wedge$$

$$c = a + s \wedge i = s \wedge i \leq b + 1;$$

Ennek birtokában elkészíthetjük az annotációs diagramot.



A következő feladat a Q-diagram alapján a verifikációs feltételek felírása és azok bizonyítása.

VERIFIKÁCIÓS FELTÉTELEK

Tétel$_1$:

$$((\varphi(a) = \text{zerus} \vee \varphi(a) = \text{succ}(n)) \wedge a \in \text{nat}_c \wedge b \in \text{nat}_c) \Rightarrow I(a, b, n, a, 0, 0);$$

azaz:

$$((\varphi(a) = \text{zerus} \vee \varphi(a) = \text{succ}(n)) \wedge a \in \text{nat}_c \wedge b \in \text{nat}_c) \Rightarrow$$

$$((\varphi(a) = \text{zerus}) \vee \varphi(a) = \text{succ}(\varphi(n))) \wedge a \in \text{nat}_c \wedge b \in \text{nat}_c) \wedge$$

$$a = a + 0 \wedge 0 = 0 \wedge 0 \leq b + 1);$$

Tétel$_2$:

$$(I(a, b, n, c, s, i) \wedge s \neq b) \Rightarrow I(a, b, n, c + 1, s + 1, i + 1);$$

azaz:

$$(((\varphi(a) = zerus) \vee \varphi(a) = succ(\varphi(n))) \wedge a \in nat_c \wedge b \in nat) \wedge$$
$$c = a + s \wedge i = s \wedge i \leq b + 1 \wedge s \neq b) \Rightarrow$$
$$((\varphi(a) = zerus) \vee \varphi(a) = succ(\varphi(n))) \wedge a \in nat_c \wedge b \in nat) \wedge$$
$$(c + 1) = a + (s + 1) \wedge (i + 1) = (s + 1) \wedge (i + 1) \leq b + 1);$$

Tétel$_3$:

$$I(a, b, n, c, s, i) \Rightarrow i \leq b + 1;$$

azaz

$$(((\varphi(a) = zerus) \vee \varphi(a) = succ(\varphi(n))) \wedge a \in nat_c \wedge b \in nat) \wedge$$
$$c = a + s \wedge i = s \wedge i \leq b + 1) \Rightarrow i \leq b + 1;$$

Tétel$_4$:

$$I(a, b, n, c, s, i) \wedge s = b \Rightarrow ((\varphi(a) = zerus \wedge \varphi(c) = \varphi(b)) \vee$$
$$(\varphi(a) = succ(n) \wedge \varphi(c) = succ(add(n, \varphi(b))) \wedge c \in nat_c)$$

azaz

$$(((\varphi(a) = zerus) \vee \varphi(a) = succ(\varphi(n))) \wedge a \in nat_c \wedge b \in nat) \wedge$$
$$c = a + s \wedge i = s \wedge i \leq b + 1) \wedge s = b) \Rightarrow$$
$$((\varphi(a) = zerus \wedge \varphi(c) = \varphi(b) \vee (\varphi(a) = succ(n) \wedge$$
$$\varphi(c) = succ(add(n, \varphi(b))) \wedge c \in nat_c)$$

Minden verifikációs feltétel igaz , a Q-diagram tehát induktív, invariáns és konzisztens.

6.4. A HOARE-MÓDSZER

A Hoare-módszer [HOA69]lényege, hogy a matematikai logikában tételek bizonyítására használatos deduktív módszert alkalmazza a programok helyességének bizonyítására.

A dedukciós tételbizonyítási módszer a következő sémával jellemezhető:

dedukció: axiómák + következtetési szabályok $\Rightarrow$ tétel

Itt tehát arról van szó, hogy a programok helyességére vonatkozó tételek az axiómákból következtetési szabályok segítségével bebizonyíthatók. Az axiómákat és a következtetési szabályokat a nyelv szemantikája alapján származtathatjuk.

AXIÓMÁK

- A **skip** üres utasítás axiómája:

$$\{P(x, y)\}\ \textbf{skip}\ \{P(x, y)\}$$

- Az értékadás axiómája:

$$\{P(x, g(x, y))\}\ y \leftarrow g(x, y)\ \{P(x, y)\}$$

KÖVETKEZTETÉSI SZABÁLYOK

- Az S_1; S_2 szekvencia következtetési szabálya:

$$\frac{\{P\}\ S_1;\ \{Q_1\}\ \textbf{és}\ \{Q_1\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2\ \{Q\}}$$

- Az **if** α **then** S_1 **else** S_2 **fi** feltételes elágazás következtetési szabálya:

$$\frac{\{P \wedge \alpha\}\ S_1\ \{Q\}\ \textbf{és}\ \{P \wedge \neg\alpha\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ \alpha\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{Q\}}$$

- A **while** α **do** S **od** iteráció következtetési szabálya:

$$\frac{\{P \wedge \alpha\}\ S\ \{P\}\ \textbf{és}\ P \wedge \neg\alpha \Rightarrow Q}{\{P\}\ \textbf{while}\ \alpha\ \textbf{do}\ S\ \textbf{od}\ \{Q\}}$$

- A következmény szabálya:

$$\frac{P \Rightarrow P_1\ \textbf{és}\ \{P_1\}\ S\ \{Q_1\}\ \textbf{és}\ Q_1 \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

TOVÁBBI SZABÁLYOK

- Az értékadás következtetési szabálya az axióma és a következmény szabályának felhasználásával a következő formára hozható:

$$\frac{P(x, y) \Rightarrow Q(x, g(x, y))}{\{P(x, y)\}\ y \leftarrow g(x, y)\ \{Q(x, y)\}}$$

- A szekvencia következtetési szabályának egy általános formája a következmény szabályának felhasználásával:

$$\frac{P \Rightarrow P_1 \textbf{ és } \{P_1\}\ S_1\ \{Q_1\} \textbf{ és } Q_1 \Rightarrow P_2 \textbf{ és } \{P_2\}\ S_2\ \{Q_2\} \textbf{ és } Q_2 \Rightarrow Q}{\{P\}\ S_1; S_2\ \{Q\}}$$

- A feltételes elágazás következtetési szabályának általános formája:

$$\frac{P \Rightarrow P_1 \textbf{ és } \{P_1 \wedge \alpha\}\ S_1\ \{Q_1\} \textbf{ és } \{P_1 \wedge \sim\alpha\}\ S_2\ \{Q_1\} \textbf{ és } Q_1 \Rightarrow Q}{\{P\}\ \textbf{if } \alpha \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\ \{Q\}}$$

- A feltételes elágazás következtetési szabálya speciális esetben:

$$\frac{P \Rightarrow P_1 \textbf{ és } \{P_1 \wedge \alpha\}\ S\ \{Q_1\} \textbf{ és } P_1 \wedge \sim\alpha \Rightarrow Q_1 \textbf{ és } Q_1 \Rightarrow Q}{\{P\}\ \textbf{if } \alpha \textbf{ then } S \textbf{ fi}\ \{Q\}}$$

- Az iterációnál bevezetve a ciklus invariánsát a következmény szabályának felhasználásával kapjuk a következő általános formát:

$$\frac{\begin{array}{c} P(x, y) \Rightarrow I(x, y) \textbf{ és} \\ \{I(x, y) \wedge \alpha(x, y)\}\ S\ \{I(x, y)\} \textbf{ és} \\ I(x, y) \wedge \sim\alpha(x, y) \Rightarrow Q(x, y) \end{array}}{\{P(x, y)\}\ \text{while } \alpha(x, y) \text{ do } S \text{ od}\ \{Q(x, y)\}}$$

- A teljes helyesség bizonyításának következtetési szabálya.

$$P(x, y) \Rightarrow I(x, y) \textbf{ és } I(x, y) \Rightarrow E(x, y) \in W_{<} \textbf{ és}$$
$$\{\{I(x, y) \wedge \alpha(x, y) \wedge E = E(x, y)\}\} \; S \; \{\{I(x, y) \wedge E(x, y) < E\}\} \textbf{ és}$$
$$I(x, y) \wedge \neg\alpha(x, y) \Rightarrow Q(x, y)$$
$$\overline{\{\{P(x, y)\} \textbf{ while } \alpha(x, y) \textbf{ do } S \textbf{ od } \{\{Q(x, y)\}\}}$$

- Az iteráció következtetési szabályának gyakran azt a formáját használjuk, amikor a W a nem negatív egész számok halmaza és $E(x, y) = i$, ami a ciklusszámlálót jelenti. Ekkor a következtetési szabály:

$$P(x, y) \Rightarrow I(x, y, 0) \textbf{ és } I(x, y, i) \Rightarrow i < k(x) \textbf{ és}$$
$$\{\{I(x, y, i) \wedge \alpha(x, y)\}\} \; S \; \{\{I(x, y, i + 1)\}\} \textbf{ és}$$
$$I(x, y, i) \wedge \neg\alpha(x, y) \Rightarrow Q(x, y)$$
$$\overline{\{\{P(x, y)\}\} \textbf{ while } \alpha(x, y) \textbf{ do } S \textbf{ od } \{\{Q(x, y)\}\}}$$

6.4.2. PÉLDA. Bizonyítsuk be a legnagyobb közös osztó kiszámításának algoritmusára vonatkozó tétel teljes helyességét a Hoare-módszerrel. A tétel a következő:

$$\{x_1 > 0 \wedge x_2 > 0\}$$
$$(y_1, y_2) \leftarrow (x_1, x_2);$$
$$\textbf{while } y_1 \neq y_2 \textbf{ do}$$
$$\textbf{if } y_1 > y_2 \textbf{ then } y_1 \leftarrow y_1 - y_2 \textbf{ else } y_2 \leftarrow y_2 - y_1 \textbf{ fi};$$
$$\textbf{od};$$
$$\{y_1 = \text{lnko}(x_1, x_2)\}$$

A ciklusinvariáns elhelyezése és végigvezetése a programon:

$\{x_1 > 0 \wedge x_2 > 0\}$

$(y_1, y_2) \leftarrow (x_1, x_2);$

$\{I(x, y, 0)\}$

$\{I(x, y, i)\}$**while** $y_1 \neq y_2$ **do**

$\{I(x, y, i) \wedge y_1 \neq y_2\}$

if $y_1 > y_2$ **then** $y_1 \leftarrow y_1 - y_2$ **else** $y_2 \leftarrow y_2 - y_1$ **fi**;

$\{I(x, y, i + 1)\}$

od;

$\{y_1 = \text{lnko}(x_1, x_2)\}$

Írjuk fel a ciklus invariánsának értékét:

$I(x, y, i)$: $y_1 > 0 \wedge y_2 > 0 \wedge \text{lnko}(y_1, y_2) = \text{lnko}(x_1, x_2) \wedge y_1 + y_2 + i \leq x_1 + x_2;$

Írjuk fel a ciklus számlálójának felső korlátját:

$$k(x_1, x_2) = x_1 + x_2;$$

Ezután következhet a tétel bizonyítása.

1. feltevés:

$$\{x_1 > 0 \wedge x_2 > 0\}\ (y_1, y_2) \leftarrow (x_1, x_2)\ \{I(x, y, 0)\}$$

2. feltevés:

$\{I(x, y, 0)\}$

while $y_1 \neq y_2$ **do**

if $y_1 > y_2$ **then** $y_1 \leftarrow y_1 - y_2$ **else** $y_2 \leftarrow y_2 - y_1$ **fi**;

od;

$\{y_1 = \text{lnko}(x_1, x_2)\}$

Ekkor a szekvencia következtetési szabálya alapján a tétel igaz:

$$\{x_1 > 0 \wedge x_2 > 0\}(y_1, y_2) \leftarrow (x_1, x_2)\{ I(x, y, 0)\} \textbf{ és}$$

$$\{I(x, y, 0)\}\textbf{while } y_1 \neq y_2 \textbf{ do}$$

$$\textbf{if } y_1 > y_2 \textbf{ then } y_1 \leftarrow y_1 - y_2 \textbf{ else } y_2 \leftarrow y_2 - y_1 \textbf{ fi};$$

$$\textbf{od}; \{y_1 = lnko(x_1, x_2)\}$$

$$\{x_1 > 0 \wedge x_2 > 0\}(y_1, y_2) \leftarrow (x_1, x_2);$$
$$\textbf{while } y_1 \neq y_2 \textbf{ do}$$

$$\textbf{if } y_1 > y_2 \textbf{ then } y_1 \leftarrow y_1 - y_2 \textbf{ else } y_2 \leftarrow y_2 - y_1 \textbf{ fi};$$

$$\textbf{od};$$

$$\{y_1 = lnko(x_1, x_2)\}$$

Most bizonyítsuk az 1. feltevést. Az értékadás következtetési szabálya alapján. $g(x_1, x_2; y_1, y_2) = (x_1, x_2)$, ezért:

$$x_1 > 0 \wedge x_2 > 0 \Rightarrow x_1 > 0 \wedge x_2 > 0 \wedge lnko(x_1, x_2) = lnko(x_1, x_2) \wedge$$

$$x_1 + x_2 + 0 \leq x_1 + x_2;$$

azaz

$$x_1 > 0 \wedge x_2 > 0 \Rightarrow I(x_1, x_2, x_1, x_2, 0)$$

$$\{x_1 > 0 \wedge x_2 > 0\}$$

$$(y_1, y_2) \leftarrow g(x, y)$$

$$\{I(x_1, x_2; y_1, y_2, 0\}$$

Bizonyítsuk a 2. feltevést.

Először be kell látni, hogy — ha az invariáns jó, akkor — biztosítva van a ciklus befejeződése: $I(x_1, x_2, y_1, y_2, i) \Rightarrow i \leq k(x_1, x_2)$, amelynek állítása triviális.

Ezután tegyünk egy újabb feltevést:

3. feltevés:

$$\{I(x_1, x_2, y_1, y_2, i) \wedge y_1 \neq y_2\}$$

if $y_1 > y_2$ **then** $y_1 \leftarrow y_1 - y_2$ **else** $y_2 \leftarrow y_2 - y_1$ **fi**;

$$\{I(x_1, x_2, y_1, y_2, i + 1)\}.$$

Végül bizonyítsuk, hogy kilépéskor teljesül a feltétel:

$$(I(x_1, x_2, y_1, y_2, i) \wedge y_1 = y_2) \Rightarrow y_1 = \text{lnko}(x_1, x_2);$$

Tehát az iteráció következtetési szabálya alapján igaz a tétel:

$$\{I(x_1, x_2, y_1, y_2, i) \wedge y_1 \neq y_2\}$$

if $y_1 > y_2$ **then** $y_1 \leftarrow y_1 - y_2$ **else** $y_2 \leftarrow y_2 - y_1$ **fi**;

$$\{I(x_1, x_2, y_1, y_2, i + 1)\}$$ **és**

$$(I(x_1, x_2, y_1, y_2, i) \wedge y_1 = y_2) \Rightarrow y_1 = \text{lnko}(x_1, x_2)$$ **és**

$$I(x_1, x_2, y_1, y_2, i) \Rightarrow i \leq k(x_1, x_2)$$

$\{I(x_1, x_2, y_1, y_2, 0)\}$**while** $y_1 \neq y_2$ **do**

if $y_1 > y_2$ **then** $y_1 \leftarrow y_1 - y_2$ **else** $y_2 \leftarrow y_2 - y_1$ **fi**;

od; $\{I(x_1, x_2, y_1, y_2, i) \wedge y_1 = y_2\}$

ha igaz a 3. feltevésünk. Bizonyítsuk be tehát ezt is, amit a feltételes elágazás következtetési szabálya alapján tehetünk meg.

$$I(x_1, x_2, y_1, y_2, i) \wedge y_1 > y_2 \Rightarrow I(x_1, x_2, y_1 - y_2, y_2, i + 1)$$ **és**

$$I(x_1, x_2, y_1, y_2, i) \wedge y_2 > y_1 \Rightarrow I(x_1, x_2, y_1, y_2 - y_1, i + 1)$$

$$\{I(x_1, x_2, y_1, y_2, i) \wedge y_1 \neq y_2\}$$

if $y_1 > y_2$ **then** $y_1 \leftarrow y_1 - y_2$ **else** $y_2 \leftarrow y_2 - y_1$ **fi**;

$$\{I(x_1, x_2, y_1, y_2, i + 1)\}$$

Vegyük észre, hogy a Hoare-módszer alkalmazása során a példánkban a következtetési szabályok alkalmazásán túl a következő tételeket bizonyítottuk:

T_1. tétel:

$$x_1 > 0 \wedge x_2 > 0 \Rightarrow I(x_1, x_2, x_1, x_2, 0);$$

azaz

$$(x_1 > 0 \wedge x_2 > 0) \Rightarrow (x_1 > 0 \wedge x_2 > 0 \wedge lnko(x_1, x_2) = lnko(x_1, x_2) \wedge$$
$$x_1 + x_2 + 0 \leq x_1 + x_2);$$

T_2. tétel:

$$(I(x_1, x_2, y_1, y_2, i) \wedge y_1 = y_2) \Rightarrow y_1 = lnko(x_1, x_2);$$

azaz

$$(y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge y_1 + y_2 + i \leq x_1 + x_2 \wedge y_1 = y_2)$$
$$\Rightarrow y_1 = lnko(x_1, x_2);$$

T_3. tétel:

$$I(x_1, x_2, y_1, y_2, i) \Rightarrow i \leq k(x_1, x_2);$$

azaz

$$(y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge$$
$$y_1 + y_2 + i \leq x_1 + x_2) \Rightarrow i \leq x_1 + x_2;$$

T_4. tétel:

$$(I(x_1, x_2, y_1, y_2, i) \wedge y_1 > y_2) \Rightarrow I(x_1, x_2, y_1 - y_2, y_2, i + 1);$$

azaz

$$(y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge$$
$$y_1 + y_2 + i \leq x_1 + x_2 \wedge y_1 > y_2) \Rightarrow$$
$$(y_1 - y_2 > 0 \wedge y_2 > 0 \wedge lnko(y_1 - y_2, y_2) = lnko(x_1, x_2) \wedge$$
$$y_1 - y_2 + y_2 + i + 1 \leq x_1 + x_2);$$

T_5. tétel:

$$(I(x_1, x_2, y_1, y_2, i) \wedge y_2 > y_1) \Rightarrow I(x_1, x_2, y_1, y_2 - y_1, i + 1)$$

azaz

$$(y_1 > 0 \wedge y_2 > 0 \wedge lnko(y_1, y_2) = lnko(x_1, x_2) \wedge$$

$$y_1 + y_2 + i \leq x_1 + x_2 \wedge 2 > y_1) \Rightarrow$$

$$(y_1 > 0 \wedge y_2 - y_1 > 0 \wedge lnko(y_1, y_2 - y_1) = lnko(x_1, x_2) \wedge$$

$$y_1 + y_2 - y_1 + i + 1 \leq x_1 + x_2);$$

Tehát a bizonyítás érdemi része az értékadásokra vonatkozó tételek bizonyítása, amely tételek megegyeznek a Floyd-módszer alkalmazása során bizonyítandó tételekkel.

6.4.2. PÉLDA Tekintsük a szorzás műveletének absztrakt leírását jelentő algebrai axiómákat. Szintaxis:

$$\times: \text{nat nat} \rightarrow \text{nat [infix]};$$

Szemantika:

$$a, c \in \text{nat};$$

$$a \times 0 = 0,$$

$$a \times c = c \times a,$$

$$a \times (c + 1) = (a \times c) + a;$$

Legyen az $m = a \times b$ szorzás algoritmusa a következő:

Q_{mp_c}:

(n, m) ← (a, 0);

while n ≠ 0 **do** s ← 0;

while s ≠ b **do** (m, s) ← (m + 1, s + 1) **od**;

n ← n – 1 **od**;

Bizonyítsuk be az algoritmus absztrakt specifikáció szerinti helyességét.

MEGOLDÁS Hozzuk először az axiómákat elő- és utófeltételes formára:

Előfeltétel:

$$\{y = 0 \vee y = c + 1\}$$

Utófeltétel:

$$\{(y = 0 \wedge z = 0) \vee (y = (c + 1) \wedge z = (a \times c) + c)\};$$

Egészítsük ki ezeket az algoritmus bemenő és eredményadatainak invariánsaival is:

$$preQ_{mp_c}: (y = 0 \vee y = c + 1) \wedge (a, c \in nat \wedge b = y);$$

$$postQ_{mp_c}: ((b = 0 \wedge z = 0) \vee (b = c + 1 \wedge z = (a \times c) + a)) \wedge (a, c \in nat);$$

Vegyük észre, hogy az algoritmus bemenő adataira kapott feltétel a következő:

$$preQ_{mp_c}(a, b): a \geq 0 \wedge b \geq 0.$$

Az eredményre pedig

$$postQ_{mp_c}(a, b, z): z = a \times b.$$

Írjuk fel mindkét ciklus invariánsát, felhasználva az "×" absztrakt műveletet!

A külső ciklus invariánsa:

$$I_k(a, b, n, m, i): m = (a - n) \times b \wedge i = a - n \wedge 0 \leq n \leq a;$$

A belső ciklus invariánsa:

$$I_b(a, b, n, m, s, i, j): m = (a - n) \times b + s \wedge i = a - n \wedge$$

$$j = s \wedge 0 \leq n \leq a \wedge 0 \leq s \leq b;$$

Helyezzük el a programban az invariánsokat.

$\{n = a \wedge m = 0 \wedge a \geq 0 \wedge b \geq 0\}$

$\{I_k(a, b, n, m, 0)\}$

$\{I_k(a, b, n, m, i)\}$ **while** $n \neq 0$ **do**

$\{I_k(a, b, n, m, i) \wedge n \neq 0\}$

$s \leftarrow 0;\{I_b(a, b, n, m, s, i, 0)\}$

$\{I_b(a, b, n, m, s, i, j)\}$ **while** $s \neq b$ **do**

$\{I_b(a, b, n, m, s, i, j) \wedge s \neq b\}$

$(m, s) \leftarrow (m + 1, s + 1)$

$\{I_b(a, b, n, m, s, i, j + 1)\}$ **od;**

$\{I_k(a, b, n, m, i)\}$

$n \leftarrow n - 1$

$\{I_k(a, b, n, m, i + 1)\}$ **od;**

$\{m = a \times b\}$

Írjuk fel mindkét ciklusszámlálóra a korlátot.

$$k_i(a, b) = a;$$

$$k_j(a, b) = b;$$

Ezután a bizonyítandó tételek a következők:

Tétel$_1$: $(n = a \wedge m = 0 \wedge a \geq 0 \wedge b \geq 0) \Rightarrow I_k(a, b, n, m, 0)$; azaz

$(n = a \wedge m = 0 \wedge a \geq 0 \wedge b \geq 0) \Rightarrow (m = (a - n) \times b \wedge i = a - n \wedge 0 \leq n \leq a)$;

Tétel$_2$: $\{ I_k(a, b, n, m, 0)\}\ S_k\ \{m = a \times b)\}$;

Az ehhez bizonyítandó tételek:

Tétel$_{21}$: $(I_k(a, b, n, m, i) \wedge n = 0 \Rightarrow m = a \times b$; azaz

$(m = (a - n) \times b \wedge i = a - n \wedge 0 \leq n \leq a \wedge n = 0) \Rightarrow m = a \times b$;

Tétel$_{22}$: $(I_k(a, b, n, m, i) \Rightarrow i \leq k_i(a, b)$; azaz

$(m = (a - n) \times b \wedge i = a - n \wedge 0 \leq n \leq a) \Rightarrow i \leq a;$

Tétel$_{23}$: $(I_k(a, b, n, m, i) \wedge n \neq 0 \wedge s = 0) \Rightarrow I_b(a, b, n, m, s, i, 0)$; azaz

$(m = (a - n) \times b \wedge i = a - n \wedge 0 \leq n \leq a \wedge n \neq 0 \wedge s = 0) \Rightarrow$

$(m = (a - n) \times b + 0 \wedge i = a - n \wedge 0 = 0 \wedge 0 \leq n \leq a \wedge 0 \leq 0 \leq b);$

Tétel$_{24}$: $(I_b(a, b, n, m, s, i, j) \wedge s = b \wedge n' = n - 1) \Rightarrow I_k(a, b, n', m, i + 1)$; azaz

$(m = (a - n) \times b + s \wedge i = a - n \wedge j = s \wedge 0 \leq n \leq a \wedge 0 \leq s \leq b \wedge$

$s = b \wedge n' = n - 1) \Rightarrow m = (a - n') \times b \wedge i + 1 = a - n' \wedge 0 \leq n' \leq a);$

Tétel$_{25}$: $I_b(a, b, n, m, s, i, j) \Rightarrow j \leq k_j(a, b)$; azaz

$(m = (a - n) \times b + s \wedge i = a - n \wedge j = s \wedge 0 \leq n \leq a \wedge 0 \leq s \leq b) \Rightarrow j \leq b;$

Tétel$_{26}$: $(I_b(a, b, n, m, s, i, j) \wedge s \neq b) \Rightarrow I_b(a, b, n, m + 1, s + 1, i, j + 1)\}$; azaz

$(m = (a - n) \times b + s \wedge i = a - n \wedge j = s \wedge 0 \leq n \leq a \wedge 0 \leq s \leq b \wedge s \neq b) \Rightarrow$

$(m + 1 = (a - n) \times b + s + 1 \wedge i = a - n \wedge$

$j + 1 = s + 1 \wedge 0 \leq n \leq a \wedge 0 \leq s + 1 \leq b)$

6.5. Strukturált programok esetén a Floyd-módszer és a Hoare-módszer ekvivalens

Hasonlítsuk össze a két módszert a determinisztikus programstruktúrák eseteiben.

1. Az értékadás esete.
   - a Hoare-módszer alkalmazása:

$$\frac{P(x, y) \Rightarrow Q(x, g(x, y))}{\{P(x, y)\}\ y \leftarrow g(x, y)\ \{Q(x, y)\}}$$

a Hoare-módszer esetén a bizonyítandó tétel tehát

$$P(x, y) \Rightarrow Q(x, g(x, y)).$$

- a Floyd-módszer alkalmazása:

$$R(x, y): \text{"true"};\ r(x, y): g(x, y);$$
$$P(x, y) \wedge \text{"true"} \Rightarrow Q(x, g(x, y))$$

a Floyd-módszer esetén ebben az esetben is ugyanarra a megoldandó feladatra jutottunk.

2. A feltételes elágazás esete.

- a Hoare-módszer alkalmazása:

$$\frac{\{P(x, y) \wedge \alpha(x, y)\}S_1\{Q(x, y)\}\ \textbf{és}\ \{P(x, y) \wedge \sim\alpha(x, y)\}S_2\{Q(x, y)\}}{\{P\}\ \textbf{if}\ \alpha\ \textbf{then}\ S_1\ \text{else}\ S_2\ \textbf{fi}\ \{Q\}}$$

- a Floyd-módszer alkalmazása:

$$R_1(x, y): \alpha(x, y);\ r_1(x, y) = f_{S_1}(x, y);$$
$$R_2(x, y): \sim\alpha(x, y);\ r_2(x, y) = f_{S_2}(x, y);$$
$$P(x, y) \wedge \alpha(x, y) \Rightarrow Q(x, f_{S_1}(x, y));$$
$$P(x, y) \wedge \sim\alpha(x, y) \Rightarrow Q(x, f_{S_2}(x, y));$$

Ha a Hoare-módszerben szereplő S_1 és S_2 programokat a programfüggvényeikre vonatkozó értékadással írjuk fel és figyelembe vesszük az értékadásra vonatkozó következtetési szabályt, akkor a két módszer ismét azonos.

3. Az iteráció esete.

   - a Hoare-módszer alkalmazása:

$$\frac{\begin{array}{c} P(x, y) \Rightarrow I(x, y, 0) \textbf{ és } I(x, y, i) \Rightarrow i < k(x) \textbf{ és} \\ \{\{I(x, y, i) \wedge \alpha(x, y)\}\}\ S\ \{\{I(x, y, i + 1)\}\} \textbf{ és} \\ I(x, y, i) \wedge \sim\alpha(x, y) \Rightarrow Q(x, y) \end{array}}{\{\{P(x, y)\}\}\ \textbf{while}\ \alpha(x, y)\ \textbf{do}\ S\ \textbf{od}\ \{\{Q(x, y)\}\}}$$

   - a Floyd-módszer alkalmazása:

$$\begin{array}{c} P(x, y) \Rightarrow I(x, y, 0); I(x, y, i) \Rightarrow i < k(x); \\ R_1(x, y): \alpha(x, y); r_1(x, y) = f_S(x, y); \\ I(x, y, i) \wedge \alpha(x, y) \Rightarrow I(x, f_S(x, y), i + 1); \\ R_2(x, y): \sim\alpha(x, y); r_2(x, y) = y; \\ I(x, y, i) \wedge \sim\alpha(x, y) \Rightarrow Q(x, y); \end{array}$$

Az előbb elmondottak szerint a programfüggvényekkel dolgozva ismét azonos feladatok megoldása szerepel mindkét módszerben.

6.6. A BIZONYÍTÁSI RENDSZER MEGBÍZHATÓSÁGA

Adva van a PS_{seq} bizonyítási rendszer és a {p} S {q} formula, amely a bizonyítási rendszerben levezethető. Jelölje a tételnek ezt a tulajdonságát:

$$\{p\}\ S\ \{q\}/PS_{seq} = \text{"true"}$$

Legyen adott a rendszer {p} S {q} = "true" formulája. Jelölje a formulának ezt a tulajdonságát:

$$\{p\}\ S\ \{q\} = \text{"true"}/PS_{seq}$$

DEFINÍCIÓ Azt mondjuk, hogy a bizonyítási rendszer **megbízható**, ha minden olyan formula, amely a rendszerben levezethető, igaz, azaz

$$\{p\}\ S\ \{q\}/PS_{seq} = \text{"true"} \Rightarrow \{p\}\ S\ \{q\} = \text{"true"}/PS_{seq}$$

A Floyd-féle indukciós állítások módszere esetén ez a következőt jelenti.

Legyen P = (L, T, s, t).

1. P programhoz készítsük el a Q állításokkal kiegészített gráfot.
2. Bizonyítsuk be, hogy az állítások diagramja **induktív**.
3. Válasszunk egy jól rendezett (W, <) halmazt, $\forall l \in L$-hez egy ρ_l: $\Sigma \rightarrow W$ függvényt.
   - Bizonyítsuk be, hogy
     $$(\forall l \in L)\ (Q_l(\sigma) = \text{"true"} \Rightarrow \rho_l(\sigma) \in W).$$
   - Bizonyítsuk be, hogy
     $$(\forall (l, \alpha \rightarrow r, l') \in T)(Q_l(\sigma) \wedge \alpha(\sigma) = \text{"true"} \Rightarrow \rho_{l'}(r(\sigma)) < \rho_l(\sigma).$$

- Bizonyítsuk be, hogy a Q gráf konzisztens a (φ, ψ) specifikáció szerint.

Ha mindezeket bebizonyítottuk, akkor azt bizonyítottuk be, hogy a $\{\varphi\}\ S\ \{\psi\}$ a Floyd-rendszerben levezethető, azaz $\{\varphi\}\ S\ \{\psi\}/PS_{seq} = \text{"true"}$.
A megbízhatóság szerint ebből következnie kell annak, hogy a P program a (φ, ψ) specifikáció szerint teljesen helyes. Valódi programok esetén ezt mind parciális, mind teljes helyesség esetén bizonyítottuk.

A BIZONYÍTÁSI RENDSZER TELJESSÉGE Legyen megadva a PS_{seq} bizonyítási rendszer és annak egy $\{p\}\ S\ \{q\} = \text{"true"}$ formulája:

$$\{p\}\ S\ \{q\} = \text{"true"}/PS_{seq}$$

Azt mondjuk, hogy a bizonyítási rendszer **teljes**, ha minden olyan formulája a rendszernek, amely igaz a rendszerben, levezethető, azaz

$$\{p\}\ S\ \{q\} = \text{"true"}/PS_{seq} \Rightarrow \{p\}\ S\ \{q\}/PS_{seq} = \text{"true"}$$

Ha tehát az indukciós módszer teljes, akkor minden, az adott specifikáció szerint helyes programról bebizonyítható a módszer segítségével, hogy helyes.

A Floyd-féle indukciós állítások módszere esetén ez a következőt jelenti.
Legyen P = (L, T, s, t). Tegyük fel, hogy P a (φ, ψ) specifikáció szerint **parciálisan helyes** $\{\varphi\}\ S\ \{\psi\}$. Akkor létezik a P programhoz egy olyan indukciós állításokat tartalmazó tranzakciós diagram (Q-diagram), amelyre

$$\varphi \Rightarrow Q_s,\ Q_t \Rightarrow \psi \text{ és } (\forall l \in L)((\varphi \Rightarrow Q_l) = \text{"true"}).$$

Most a Hoare-módszer megbízhatóságát bizonyítsuk be!

ÁLLÍTÁS A Hoare-módszer következtetési szabályai helyesek.

1. **Bizonyítsuk be, hogy az iteráció következtetési szabálya megbízható.** Legyen adott az

$$S(x, y): \textbf{while } \alpha(x, y) \textbf{ do } A(x, y) \textbf{ od};$$

iteráció, amelynek jelentését az

$$f_s(x, y) = \textbf{if } \sim\alpha(x, y) \textbf{ then } y \textbf{ else } f_s(x, f_A(x, y)) \textbf{ fi};$$

programfüggvénnyel definiáljuk, ahol $\alpha(x, y)$ kvantorfüggetlen logikai kifejezés.

Adva van az $S(x, y)$ iteráció jelentésének specifikációja elő- és utófeltételekkel:

$$\{P(x, y)\}\ S(x, y)\ \{R(x, y)\};$$

Legyenek továbbá adottak a következők:

- $I(x, y, i)$: ciklusinvariáns,
- $k(x)$: a ciklusszámláló felső korlátja,
- $f_A(x, y)$: a ciklusmag programfüggvénye.

Ha bebizonyítjuk, hogy

- $P(x, y) \Rightarrow I(x, y, 0)$,
- $I(x, y, i) \Rightarrow i \leq k(x)$,
- $I(x, y, i) \wedge \alpha(x, y) \Rightarrow I(x, f_A(x, y), i + 1)$,
- $I(x, y, i) \wedge \sim\alpha(x, y) \Rightarrow R(x, y)$,

akkor minden olyan y_0-ra, amelyre

$$P(x, y_0) = \text{"true"},$$

létezik az $f_s(x, y_0)$, és erre az értékre

$$R(x, f_s(x, y_0)) = \text{"true"}.$$

BIZONYÍTÁS A bizonyítást $k(x)$ szerinti teljes indukcióval végezhetjük el.

Alapeset: $k(x) = 0$. Ekkor

$$k(x) = 0 \Rightarrow \alpha(x, y_0) = \text{"false"}.$$

Tegyük fel ugyanis, hogy

$$\alpha(x, y_0) = \text{"true"},$$

akkor a fenti tételek szerint

$$I(x, f_A(x, y_0), 1) = \text{"true"},$$

és ezért

$$1 \leq k(x),$$

ami ellentmondás.

Így

$$I(x, y_0, 0) \wedge \sim\alpha(x, y_0) \Rightarrow R(x, y_0),$$

ámde most

$$f_s(x, y_0)) = y_0.$$

Indukció: Legyen $k(x) > 0$ és feltesszük, hogy $k'(x) \leq k(x) - 1$ -re az állítás igaz.

Tegyük fel, hogy

$$\alpha(x, y_0) = \text{"false"}.$$

Ekkor a fentiekhez hasonlóan belátható, hogy igaz az állítás.

Tegyük fel, hogy

$$\alpha(x, y_0) = \text{"true"}.$$

Ekkor

$$I(x, f_A(x, y_0), 1) = \text{"true"}.$$

Legyen tehát

$$y_1 = f_A(x, y_0),$$

$$I_1(x, y_1, i) = I(x, f_A(x, y_0), i + 1).$$

Ekkor az új ciklusszámláló korlátjára:

$$I_1(x, y_1, i) \Rightarrow i \leq k(x) - 1.$$

y_1-re tehát létezik $f_s(x, y_1)$, és erre

$$R(x, fs(x, y_1)) = \text{"true"},$$

azaz

$$R(x, f_s(x, f_A(x, y_0)) = \text{"true"},$$

Ámde most

$$f_s(x, f_A(x, y_0)) = f_s(x, y_0).$$

Ezzel az iteráció következtetési szabályának helyességét bebizonyítottuk.

2. **Tekintsük most a feltételes elágazás esetét.** Legyen adott az

$$S(x, y): \textbf{if } \alpha(x, y) \textbf{ then } A(x, y) \textbf{ else } B(x, y) \textbf{ fi};$$

amelynek jelentése:

$$f_s(x, y) = \textbf{if } \alpha(x, y) \textbf{ then } f_A(x, y) \textbf{ else } f_B(x, y) \textbf{ fi}.$$

Adva van az S(x, y) feltételes elágazás specifikációja:

$$\{P(x, y)\}\ S(x, y)\ \{R(x, y)\}$$

elő- és utófeltételekkel.

Ha bebizonyítjuk, hogy

$$P(x, y) \wedge \alpha(x, y) \Rightarrow R(x, f_A(x, y)),$$

$$P(x, y) \wedge \sim\alpha(x, y) \Rightarrow R(x, f_B(x, y)),$$

akkor minden olyan y_0-ra, amelyre $P(x, y_0) = \text{"true"}$, létezik $f_s(x, y_0)$ és az kielégíti az R relációt:

$$R(x, f_s(x, y_0)).$$

BIZONYÍTÁS Az állítás

$$f_s(x, y) = \textbf{if } \alpha(x, y) \textbf{ then } f_A(x, y) \textbf{ else } f_B(x, y) \textbf{ fi;}$$

alapján triviális.

Hasonlóan látható be a szekvencia következtetési szabályának helyessége is.

6.7. A HOARE-MÓDSZER TELJESSÉGI TÉTELE

Legyen adva az S(x, y) strukturált program, amelynek egy tetszőleges részprogramját s(x, y) jelöli.

Tegyük fel, hogy igaz a következő tétel minden ilyen s(x, y) részprogramra:

$$\{Q(x) \wedge y = I_s(x)\}\ s(x, y)\ \{Q(x) \wedge y = O_s(x)\},$$

ahol

$$f_s(x, I_s(x)) = O_s(x),$$

azaz $I_s(x)$ jelöli az s(x, y) függvény bemenő és $O_s(x)$ a hozzá tartozó eredmény adatát a program x bemenő paraméterei mellett.

ÁLLÍTÁS Minden ilyen tulajdonságú részprogramra vonatkozó fenti tétel, a Hoare-módszer segítségével bebizonyítható.

MEGJEGYZÉS Mivel definíció szerint

$$\{Q(x) \wedge y = x\}\ S(x, y)\ \{Q(x) \wedge y = O_s(x)\},$$

azaz

$$P(x): Q(x) \wedge y = x; \text{ és } R(x, y): Q(x) \wedge y = O_s(x)$$

Így az állítás magára az S(x, y) strukturált programra is vonatkozik a P(x), R(x, y) specifikáció mellett.

MEGJEGYZÉS Mi most csak a parciális helyességet bizonyítjuk, a teljes helyesség bizonyítása hasonlóan végezhető el.

BIZONYÍTÁS A bizonyítást az alapstruktúrák egymásba skatulyázásának száma szerinti teljes indukcióval hajtjuk végre.

Alapeset: Az értékadás esete:

$$s(x, y) = y \leftarrow g(x, y).$$

A tétel tehát:

$$\{Q(x) \wedge y = I_s(x)\}\ y \leftarrow g(x, y)\ \{Q(x) \wedge y = O_s(x)\},$$

amelyre az értékadás következtetési szabálya alapján azt kell belátni, hogy

$$Q(x) \wedge y = I_s(x) \Rightarrow Q(x) \wedge g(x, y) = O_s(x),$$

ami definíció szerint igaz.

$$\frac{Q(x) \wedge y = I_s(x) \Rightarrow Q(x) \wedge g(x, y) = O_s(x)}{\{Q(x) \wedge y = I_s(x)\}\ y \leftarrow g(x, y)\ \{Q(x) \wedge y = O_s(x)\}}$$

Indukció: Tegyük fel, hogy "k" mélységű egymásba skatulyázás esetén a tétel igaz, és bizonyítsuk "k+1"-re is. Három esetet kell megvizsgálni:

a. a k + 1-edik struktúra egy szekvencia:

$$s(x, y) = S_1(x, y);\ S_2(x, y);$$

Legyenek a programfüggvények rendre:

$$f_{s_1}(x, y),\ f_{s_2}(x, y).$$

Ekkor indukciós feltevésünk szerint a

$$\{Q(x) \wedge y = I_{s_1}(x)\}\ S_1(x, y)\ \{Q(x) \wedge y = O_{s_1}(x)\},$$

$$\{Q(x) \wedge y = I_{s_2}(x)\}\ S_2(x, y)\ \{Q(x) \wedge y = O_{s_2}(x)\},$$

tételek helyessége a Hoare-módszerrel bebizonyítható,

azaz

$$O_{s_1}(x) = f_{s_1}(x, I_{s_1}(x)),\ \text{és}\ O_{s_2}(x) = f_{s_2}(x, I_{s_2}(x)).$$

Ámde a szekvencia szemantikája alapján

$$I_s(x) = I_{s_1}(x),\ \text{és}\ O_{s_1}(x) = I_{s_2}(x)\ \text{és}\ O_{s_2}(x) = O_s(x);$$

ezért

$$f_s(x, I_s(x)) = f_{s_2}(x, f_{s_1}(x, I_{s_1}(x))),$$

tehát

$$\frac{\{Q(x) \wedge y = I_{s_1}(x)\}\ S_1(x, y)\ \{Q(x) \wedge y = O_{s_1}(x)\},\quad \{Q(x) \wedge y = I_{s_2}(x)\}\ S_2(x, y)\ \{Q(x) \wedge y = O_{s_2}(x)\},}{\{Q(x) \wedge y = I_s(x)\}\ S_1(x, y);\ S_2(x, y)\ \{Q(x) \wedge y = O_s(x)\},}$$

b. a k + 1-edik struktúra egy feltételes elágazás:

$$s(x, y) = \textbf{if } \alpha(x, y) \textbf{ then } S_1(x, y) \textbf{ else } S_2(x, y) \textbf{ fi} ;$$

Ekkor indukciós feltevésünk szerint a

$$\{Q(x) \wedge y = I_{s_1}(x)\}\ S_1(x, y)\ \{Q(x) \wedge y = O_{s_1}(x)\},$$

$$\{Q(x) \wedge y = I_{s_2}(x)\}\ S_2(x, y)\ \{Q(x) \wedge y = O_{s_2}(x)\},$$

tételek helyessége a Hoare-módszerrel bebizonyítható.

Ámde most $(Q(x) \wedge y = I_s(x, y) \wedge \alpha(x, y)) \Rightarrow (Q(x) \wedge y = I_{s_1}(x, y))$,

$(Q(x) \wedge y = I_s(x, y) \wedge \sim\alpha(x, y)) \Rightarrow (Q(x) \wedge y = I_{s_2}(x, y))$,

másrészt $(Q(x) \wedge y = I_{s_1}(x, y) \wedge \alpha(x, y)) \Rightarrow O_s(x) = O_{s_1}(x)$,

$(Q(x) \wedge y = I_{s_2}(x, y) \wedge \sim\alpha(x, y)) \Rightarrow O_s(x) = O_{s_2}(x)$, tehát

$$Q(x) \wedge y = I_s(x, y) \wedge \alpha(x, y)) \Rightarrow (Q(x) \wedge y = I_{s_1}(x, y)),$$

$$(Q(x) \wedge y = I_s(x, y) \wedge \sim\alpha(x, y)) \Rightarrow (Q(x) \wedge y = I_{s_2}(x, y)),$$

$$\{Q(x) \wedge y = I_{s_1}(x))\}\ S_1(x, y)\ \{Q(x) \wedge y = O_{s_1}(x)\},$$

$$\{Q(x) \wedge y = I_{s_2}(x)\}\ S_2(x, y)\ \{Q(x) \wedge y = O_{s_2}(x)\},$$

$$(Q(x) \wedge y = O_{s_1}(x)) \Rightarrow y = O_s(x),$$

$$(Q(x) \wedge y = O_{s_2}(x)) \Rightarrow y = O_s(x).$$

$$\{Q(x) \wedge y = I_s(x)\}$$

$$\textbf{if } \alpha(x, y) \textbf{ then } S_1(x, y) \textbf{ else } S_2(x, y) \textbf{ fi}$$

$$\{Q(x) \wedge y = Os(x)\},$$

c. a k + 1-edik struktúra egy iteráció:

$$s(x, y) = \textbf{while } \alpha(x, y) \textbf{ do } A(x, y) \textbf{ od},$$

ahol

$$\{Q(x) \wedge y = I_A(x)\}\ A(x, y)\ \{Q(x) \wedge y = O_A(x)\}$$

már bizonyítható a Hoare-módszerrel. Legyen A(x, y) programfüggvénye $f_A(x, y)$. Használjuk fel tehát a következő formában ezt a tételt:

$$\{Q(x) \wedge y = I_A(x)\}\ y \leftarrow f_A(x, y)\ \{Q(x) \wedge y = O_A(x)\}.$$

Vezessük be a következő jelölést:

$$h(x, y, k) = f_A(x, f_A(x, \ldots (f_A(x, y)) \ldots)), \text{ ha } k > 0;$$

$$\underbrace{1 \quad 2 \ldots \quad k}$$

$$h(x, y, k) = y, \text{ ha } k = 0.$$

Legyen az invariáns

$$I(x, y, k): Q(x) \wedge y = h(x, I_A(x), k) \wedge f_s(x, y) = f_s(x, I_s(x)).$$

amely az iteráció szemantikáját írja le.

A bizonyítandó tételek:

1. $(Q(x) \wedge y = I_s(x)) \Rightarrow I(x, y, 0)$;
2. $\{I(x, y, k) \wedge \alpha(x, y)\}\ A(x, y)\ \{I(x, y, k + 1)\}$;
3. $(I(x, y, k) \wedge \sim\alpha(x, y)) \Rightarrow (Q(x) \wedge y = O_s(x))$.

Az 1.tétel bizonyítása:

$$(Q(x) \wedge y = I_s(x)) \Rightarrow I(x, y, 0),$$

azaz

$$(Q(x) \wedge y = I_s(x)) \Rightarrow (Q(x) \wedge y = I_A(x) \wedge f_s(x, y) = f_s(x, I_s(x))),$$

ami triviális.

A 2.tétel bizonyítása:

$$\{I(x, y, k) \wedge \alpha(x, y)\}\ A(x, y)\ \{I(x, y, k + 1)\},$$

az értékadás következtetési szabálya alapján bizonyítandó:

$$(I(x, y, k) \wedge \alpha(x, y)) \Rightarrow I(x, f_A(x, y), k + 1),$$

azaz

$$(Q(x) \wedge y = h(x, I_A(x), k) \wedge f_s(x, y) = f_s(x, I_s(x)) \wedge \alpha(x, y)) \Rightarrow$$

$$(Q(x) \wedge f_A(x, y) = h(x, I_A(x), k + 1) \wedge f_s(x, f_A(x, y)) = f_s(x, I_s(x)).$$

A 3.tétel bizonyítása:

$$(I(x, y, k) \wedge \sim\alpha(x, y)) \Rightarrow (Q(x) \wedge y = O_s(x)),$$

azaz

$$(Q(x) \wedge y = h(x, I_A(x), k) \wedge f_s(x, y) = f_s(x, I_s(x)) \wedge \sim\alpha(x, y)) \Rightarrow (Q(x) \wedge y = O_s(x)).$$

Mivel $\sim\alpha(x, y)$ esetén $f_s(x, y) = y$, másrészt az

$$f_s(x, I_s(x)) = O_s(x))$$

definíció alapján a 3.tétel állítása igaz.

Írhatjuk tehát, hogy

$$Q(x) \wedge y = I_s(x) \Rightarrow I(x, y,0);$$

$$\{I(x, y, k) \wedge \alpha(x, y)\}\ A(x, y)\ \{I(x, y, k + 1)\};$$

$$I(x, y, k) \wedge \sim\alpha(x, y) \Rightarrow Q(x) \wedge y = O_s(x).$$

$$\{Q(x) \wedge y = I_s(x)\}$$

while $\alpha(x, y)$ **do** $A(x, y)$ **od**,

$$\{Q(x) \wedge y = O_s(x)\}.$$

6.8. A STRUKTURÁLT PROGRAMOZÁS ALAPTÉTELE

6.8.1. BÖHM ÉS JACOPINI TÉTELE Bármely valódi programhoz meg lehet konstruálni egy vele ekvivalens strukturált programot [BÖH66].

BIZONYÍTÁS A bizonyítást az élek száma szerinti teljes indukcióval végezzük el. Ehhez először meghatározzuk a valódi programban az élek számát a csomópontok függvényében.

Számoljuk meg a csomópontokból kifelé mutató éleket. A valódi program definíciója alapján kapjuk, hogy minden predikátum csomópontból két él mutat ki. Minden gyűjtő csomópontból, valamint függvény csomópontból pedig egy-egy

él mutat ki. Ha ezekhez hozzávesszük még az indulási csomópontba bemutató egy darab élet is, akkor megszámoltuk a valódi program gráfjában lévő összes élet. Így a valódi programban az élek száma:

$e = 2\pi + \gamma + \varphi + 1$, ahol

π = a predikátum csomópontok száma;

φ = a függvény csomópontok száma;

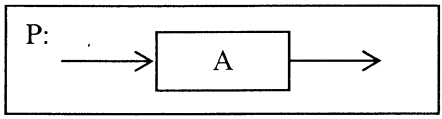
γ = a gyűjtő csomópontok száma;

A definíció alapján nyilvánvaló azonban, hogy $\pi = \gamma$, ezért

$$e = 3\pi + \varphi + 1.$$

Ezután jöhet az élek száma szerinti teljes indukció.

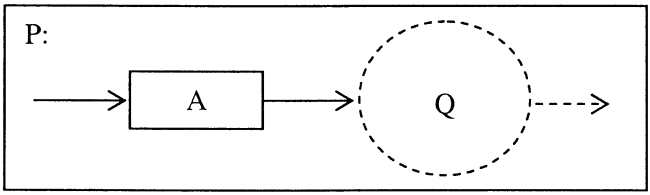
Alapeset: $e = 1$, ill. $e = 2$; $\pi = 0$; $\varphi = 1$; jelölje a függvény csomópontot A:



Az alapesetekben az állítás triviális.

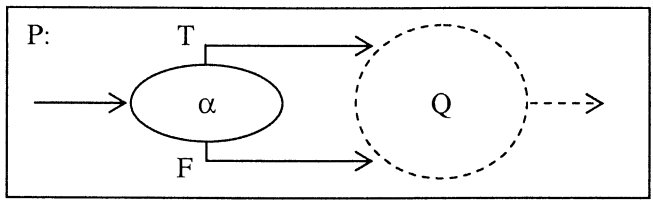
Indukció: Legyen a P programban lévő élek száma $e(P) > 2$; Feltesszük, hogy minden $e(P') < e(P)$ esetén az állításunk igaz. Vegyük sorba a lehetséges eseteket!

1. eset: Induláskor az él egy függvény csomópontba mutat:

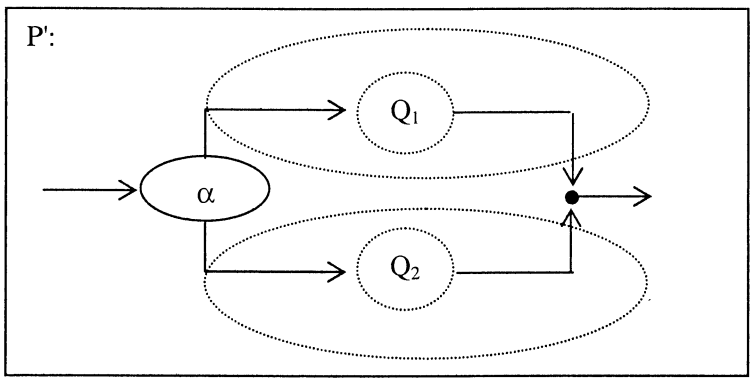


Ekkor $e(Q) = 3\pi + (\varphi - 1) + 1 < e(P)$, ezért Q-hoz meg lehet konstruálni egy vele ekvivalens Q' strukturált programot, és így a P': A; Q'; szekvencia is az eredeti P programmal ekvivalens strukturált program lesz.

2. eset: Induláskor az él egy predikátum csomópontba mutat.



Ekkor induljunk el a predikátum csomópontból az igaz ágon kiinduló utak mentén, és gyűjtsük össze az összes olyan út mentén lévő éleket a csomópontokkal együtt, amelyek a kilépő élhez vezetnek. Nyilvánvalóan ezek szintén egy valódi programot alkotnak. Jelölje ezt Q_1. Hasonló módon kaphatjuk a hamis ágon elindulva a Q_2 valódi programot is. Az eredmény a következő:



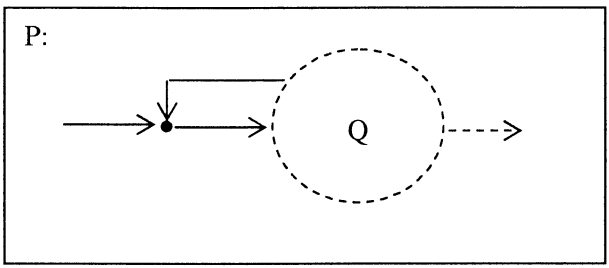
Konstrukciója alapján a P' program nyilvánvalóan ekvivalens az eredeti P programmal. Mind Q_1-ben, mind pedig Q_2-ben legalább egy predikátum csomóponttal kevesebb van, mint P-ben volt. Így ezekre alkalmazhatjuk az

indukciós feltevésünket és megkonstruálhatjuk a velük ekvivalens Q_1' és Q_2' strukturált programokat. Az eredmény a

$$P'' = \textbf{if } \alpha \textbf{ then } Q_1' \textbf{ else } Q_2' \textbf{ fi};$$

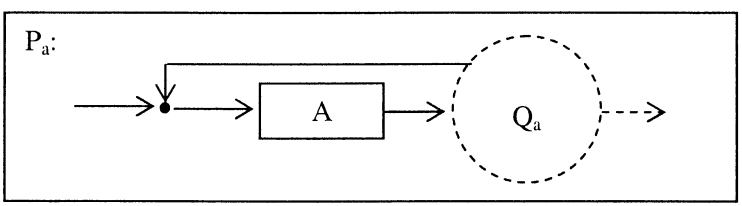
strukturált program lesz.

3. eset. Induláskor az él egy gyűjtő csomópontba mutat.

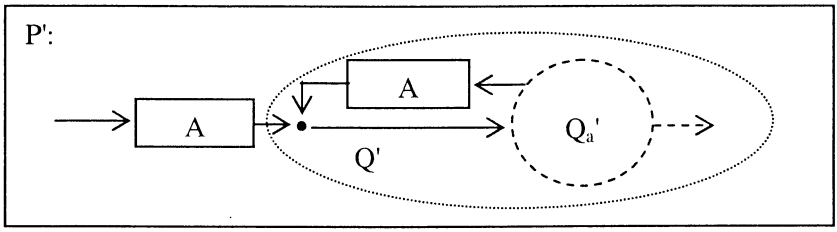


Ezen a szinten az indukciós feltevésünkkel nem tudunk még eredményt elérni. Ha itt van egy gyűjtő csomópont, akkor a programban kell lenni legalább egy predikátum csomópontnak is. Átalakításokkal ezt kívánjuk kihasználni. Vegyük tehát sorba a gyűjtő csomópontot követő csomópontok lehetséges eseteit.

3a. eset.

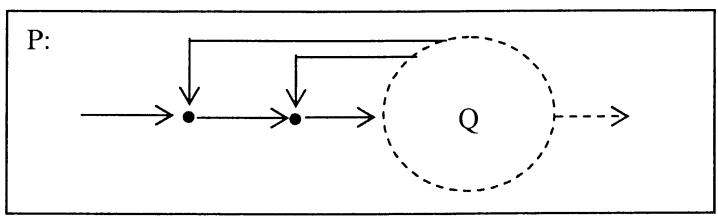


Ebből a szerkezetből hozzunk létre szekvenciát a következő módon:

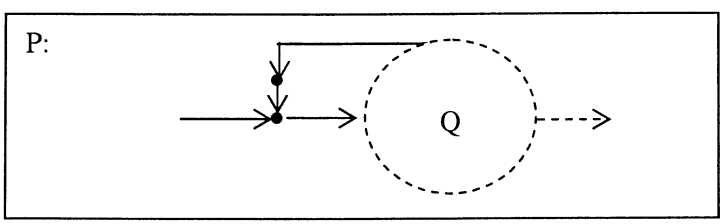


A P' program a P_a programmal ekvivalens. A Q' programban azonban az élek száma azonos a P_a programban lévő élek számával. Az indukciós feltevésünket még nem tudjuk felhasználni. Ennek az ekvivalens átalakításnak azonban az a haszna, hogy "egy lépéssel", egy "élhosszal", közelebb kerültünk a gyűjtő csomóponthoz Q_a -ban.

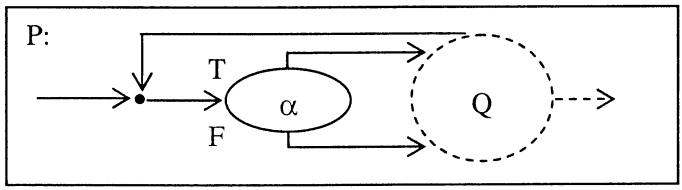
3b. eset. Nézzük most azt az esetet, amikor gyűjtő csomópontot ismét gyűjtő csomópont követ.



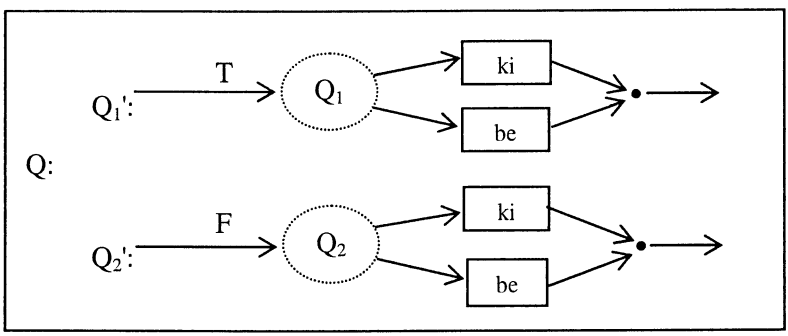
Ekkor a következő – nyilvánvalóan ekvivalens – átalakítással ismét egy éllel közelebb jutunk a predikátum csomóponthoz.



3c. eset. A probléma megoldásának kulcsa tehát az az eset, amikor a gyűjtő csomópontot predikátum csomópont követi.



Most élhetünk a Q programnak azzal az átalakításával, melyet a 2. esetben alkalmaztunk.



Itt azonban meg kell jelölni azt, hogy Q melyik éle az, amelyik kifelé és melyik, amely vissza, a gyűjtő csomópontba vezet. Ezeket szolgálja egy jelzőbitnek történő "be", ill. "ki" értékadás.

A P program éleinek száma:

$$e(P) = 3\pi + \varphi + 1.$$

A Q program éleinek száma:

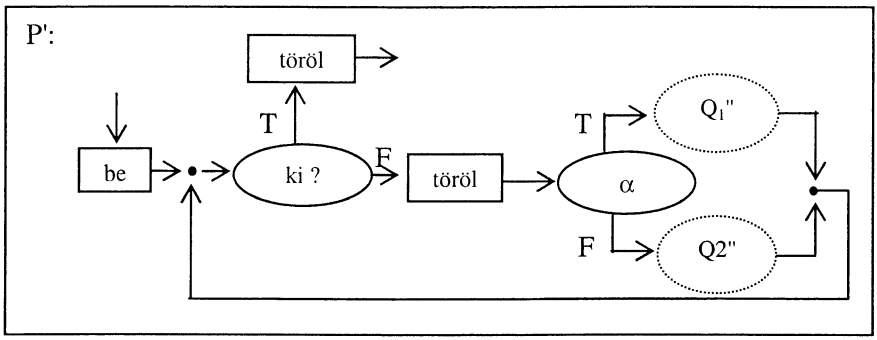
$$e(Q) = 3(\pi\text{-}1) + \varphi + 1.$$

A Q_1', Ill. Q_2' programok éleinek számára viszont a következő korlátokat kapjuk:

$$e(Q_1') \leq e(Q) + 2 = 3(\pi\text{-}1) + (\varphi+2) + 1 < e(P);$$

$$e(Q1') \leq e(Q) + 2 = 3(\pi-1) + (\varphi+2) + 1 < e(P);$$

A stukturált megoldás:



Mivel jelzőbitet vezettünk be, ezért a program megfelelő pontjaiban ezt a jelzőbitet törölni kell annak érdekében, hogy az eredeti P programmal ekvivalens eredményt kapjunk. Ezt a célt szolgálja a "töröl" operáció.

Az erdmény tehát a következő strukturált program:

P' = be; **while** ki? **do begin** töröl; **if** α **then** Q_1" **else** Q_2" **fi end od**; töröl;

Ezzel a strukturált programokra vonatkozó egzisztenciaállítást bebizonyítottuk.

7. TÍPUSÖRÖKLŐDÉS

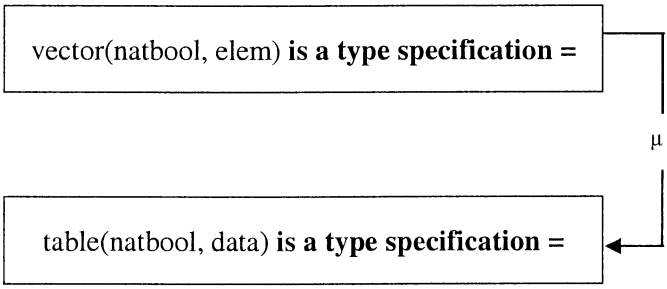
Az adattípusok specifikációi között gyakran találkozunk egymáshoz nagyon "hasonlító" leírásokkal. Ilyen esetekben ha az egyik specifikáció hibamentesen rendelkezésre áll, akkor abból előállíthatjuk a hozzá hasonló másikat. Erre szolgál a típusöröklődés, amely specifikációk közötti morfizmussal definiálható. Tekintsük például a vektor típus specifikációját.

```
vector(natbool, elem) is a type specification =
    parameters = natbool +
            sorts: elem
            oprs: ⊥: → elem
    exports =
        class sorts: vector
            oprs:
                    nil: → vector
                    put: vector natbool elem → vector
                    access: vector natbool → elem
            eqns: v∈vector; i, j∈natbool; e∈elem;
                    access(nil, i) = ⊥
                    access(put(v, i, e), j) = if i = j then e else access(v, j) fi
end vector;
```

A vektor alapvető tulajdonsága az, hogy az elemeit indexértékek azonosítják. Ehhez nagyon hasonlít a táblázat, ahol viszont az elemeket a kulcsok értékei azonosítják. Legyen tehát a táblázat specifikációja a következő:

```
table(natbool, data) is a type specification =
    parameters = natbool +
        sorts: data
        oprs: ⊥: → data
    exports =
        class sorts: table = vector
        oprs: null: → table
                assign: table nat data → table
                delete: table nat → table
                read: table nat → data
        eqns: t∈table; p, q∈nat; d∈data;
                delete(null, d) = bempty
                read(null, p) = ⊥
                delete(assign(t, p, d), q) = if p = q then delete(t, q)
                    else assign(delete (t, q), p, d) fi
                read(assign(t, p, d), q) =
                    if p = q then d else read(t, q) fi
end table;
```

A táblázat típusának specifikációja nagyon "hasonlít" a vektor típus specifikációjához. A következő ábra azt szemlélteti, hogy a táblázat specifikációját a vektor specifikációjából egyszerű morfizmussal származtathatjuk.



Ennél a származtatásnál a μ specifikációmorfizmus a következő:

$$\mu_s(\text{vector}) = \text{table};\ \mu_s(\text{natbool}) = \text{natbool};\ \mu_s(\text{elem}) = \text{data};$$

$$\mu_{op}(\bot \rightarrow \text{elem}) = \bot \rightarrow \text{data};\ \mu_{op}(\text{nil} \rightarrow \text{vector}) = \text{null} \rightarrow \text{table};\ \ldots;$$

$$\mu(v) = t;\ \mu(i) = p,\ \mu(j) = q;\ \mu(e) = d;$$

$$\mu_T(\text{access}(\text{nil}, i)) = \text{read}(\text{null}, p);\ \mu_T(\bot) = \bot;\ \ldots$$

Adattípusok specifikációjánál kétféle specifikációöröklődésről szoktunk beszélni. Ezek a következők:

- specializációval történő öröklődés.
- újrafelhasználással történő öröklődés.

7.1. SPECIALIZÁCIÓVAL TÖRTÉNŐ TÍPUSÖRÖKLŐDÉS A SPECIFIKÁCIÓ SZINTJÉN

DEFINÍCIÓ Legyenek megadva a

$$C_1 = (PAR_1, EXP_1, IMP_1, BOD_1, e_1, i_1, eb_1, ib_1)$$

és a

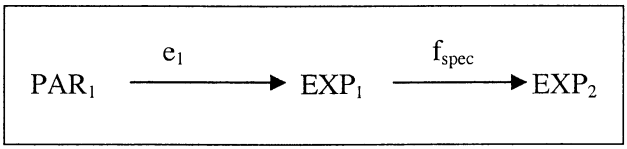
$$C_2 = (PAR_2, EXP_2, IMP_2, BOD_2, e_2, i_2, eb_2, ib_2),$$

azonos szignatúrájú típusosztályok, ahol

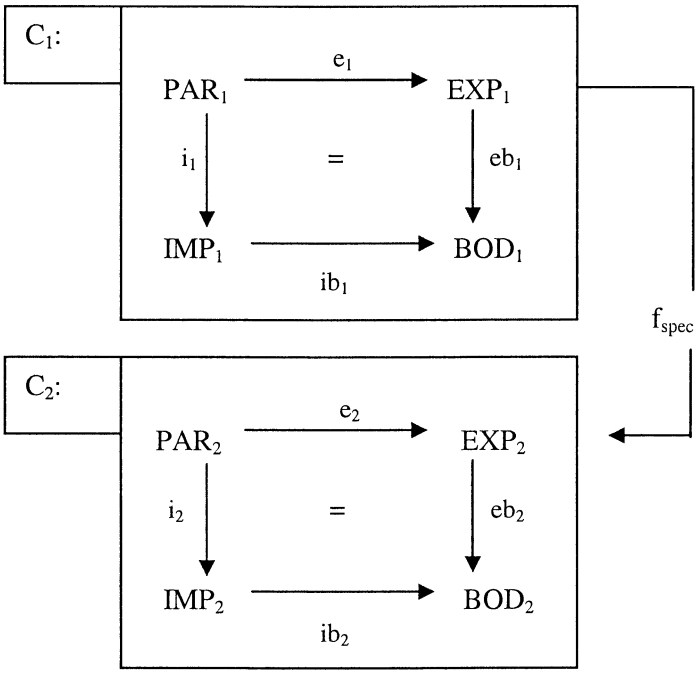
$$EXP_1=(S_1, OP_1, E_1),\ EXP_2 = (S_2, OP_2, E_2).$$

A C_2 típusosztály specifikációját a C_1 típusosztály specifikációjából specializációval történő öröklődésnek nevezzük, ha létezik egy

f_{spec}: $EXP_1 \rightarrow EXP_2$ kitüntetett elemű specifikációmorfizmus a következő ábrának megfelelően:



A specializációval történő öröklődést az osztályok morfizmusdiagramjainak feltüntetésével a következő ábra mutatja:



Tehát az öröklődés morfizmusa a következő:

Itt az f_{spec} specifikációmorfizmus a következő tulajdonságok diszjunkciója:

6. Identitás.

7. Újraelnevezés:
   - A szort új nevet kap;

     sort =

     <új név> = <régi név>;

     Például: table = vector
   - Az operáció új nevet kap;

     oprs =

     <új opr. név> = <régi opr.név>;

     Például: assign = _ [_] ::= _
8. Tartalmazás:
   - szortok esetében bővül a szortok listája;
   - operációk esetében újabb operációszimbólumokkal bővül a lista;
   - újabb axiómákkal bővül a lista, az axiómákban a változók helyén az új nevű szortok ill. új szortok példányai jelennek meg;

   Például: Legyen az absztrakt axióma:

$$v \in \text{vector}; j, k \in \text{natbool}; e \in \text{elem};$$

$$(v[j] ::= e)[k] = \textbf{if } j = k \textbf{ then } e \textbf{ else } v[k] \textbf{ fi};$$

Akkor a megfelelő konkrét axióma a következő lesz:

$$t \in \text{table}; k_1, k_2 \in \text{key}; d \in \text{data};$$

$$\text{assign} = _ [_] ::= _ ;$$

$$\text{read} = _ [_];$$

$$\text{read}(\text{assign}(t, k_1, d), k_2) = \textbf{if } k_1 = k_2 \textbf{ then } d \textbf{ else } \text{read}(t, k_2) \textbf{ fi};$$

7.1.1. PÉLDA Adott a verem specifikációja. Specializációval történő öröklődéssel állítsuk elő belőle a kettős sor specifikációját.

verem(elem, n: **natbool**) **is a class specification** =

parameters = **natbool** +

sorts: elem

oprs: $\perp$: $\rightarrow$ elem

exports=

class sorts: verem

oprs: create: $\rightarrow$ verem

push: verem elem $\rightarrow$ verem

pop: verem $\rightarrow$ verem

top: verem $\rightarrow$ elem

length: verem $\rightarrow$ natbool

eqns: v$\in$ verem; e$\in$ elem;

pop(create) = create

top(create) = $\perp$

length(create) = zerus

pop(push(v, e)) = v

top(push(v, e)) = e

length(push(v, e)) = succ(length(v))

length(v) $\geq$ n $\Rightarrow$ push(v, e) = "undefined"

imports =

body =

sorts: vector

oprs: new: $\rightarrow$ vector

put: vector natbool elem $\rightarrow$ vector

access: vector natbool $\rightarrow$ elem

lengthc: vector $\rightarrow$ natbool

rep: vector natbool $\rightarrow$ verem

eqns: v$\in$ vector; j, k, m$\in$ natbool; e$\in$ elem;

access(new, j) = ⊥

access(put(v, j, e), k) = **if** j = k **then** e

else access(v, k) **fi**

lengthc(new) = zerus

lengthc(put(v, j, e)) = succ(lengthc(v))

create = (new, zerus)

push(v, m, e) = (put(v, succ(m), e), succ(m))

pop(v, m) = (v, prec(m))

top(v, m) = access(v, m)

length(v, m) = lengthc(v)

end verem;

dque(elem, n: **natbool**) **is a specialization of verem** =

parameters = **natbool** +

sorts: elem

oprs: ⊥: → elem

exports=

class sorts: dque = verem

oprs:

init = create

addt = push

remt = pop

headt = top

addh: dque elem → dque

remh: dque → dque

headh: dque → elem

eqns: q∈dque; e, e'∈elem;

addh(init, e) = addt(init, e)

remh(init) = init

headh(init) = ⊥

addh(addt(q, e'), e) = addt(addh(q, e), e')

remh(addt(q, e)) = **if** q = init **then** init

else addt(remt(q), e) **fi**

headh(addt(q, e)) = **if** q = init **then** e **else** headh(q) **fi**

imports =

sorts: vector

oprs: new: → vector

put: vector natbool elem → vector

shiftr: vector → vector

shift*l*: vector → vector

access: vector natbool → elem

eqns: v∈ vector; j, k, m∈ natbool; e∈ elem;

access(new, j) = ⊥

access(put(v, j, e), k) = **if** j = k **then** e

else access(v, k) **fi**

shiftr(new) = new

shiftr(put(v, j, e)) = put(shiftr(v), j + 1, e)

shift*l*(new) = new

shift*l*(put(v, j, e)) = **if** j = 1 **then** shift*l*(v)

else put(shift*l*(v), j - 1, e) **fi**

body =

oprs: rep: vector natbool → dque

eqns: v∈ vector; m∈ natbool; e∈ elem;

init = (new, m)

addh(v, m, e) = **if** m < n
then (put(shiftr(v), 1, e), m + 1)
else "undefined" **fi**
remh(v, m) = **if** m = 0
then (v, m) **else** (shift*l*(v), m - 1) **fi**
headh(v, m) = **if** m = 0 **then** ⊥ **else** access(v, 1) **fi**
addt(v, m) = **if** m < n **then** (put(v, m + 1, e), m + 1)
else "undefined" **fi**
remt(v, m) = **if** m = 0 **then** (v, m) **else** (v, m - 1) **fi**
headt(v, m) = **if** m = 0 **then** ⊥ **else** access(v, 1) **fi**
length(v, m) = m

end dque;

7.2. ÚJRAFELHASZNÁLÁSSAL TÖRTÉNŐ TÍPUSÖRÖKLŐDÉS

DEFINÍCIÓ Legyenek adva a

$$C_1 = (PAR_1, EXP_1, IMP_1, BOD_1, e_1, i_1, eb_1, ib_1)$$

és a

$$C_2 = (PAR_2, EXP_2, IMP_2, BOD_2, e_2, i_2, eb_2, ib_2)$$

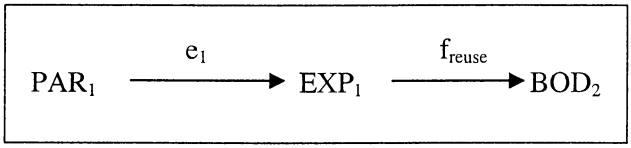
azonos szignatúrájú típusosztályok,

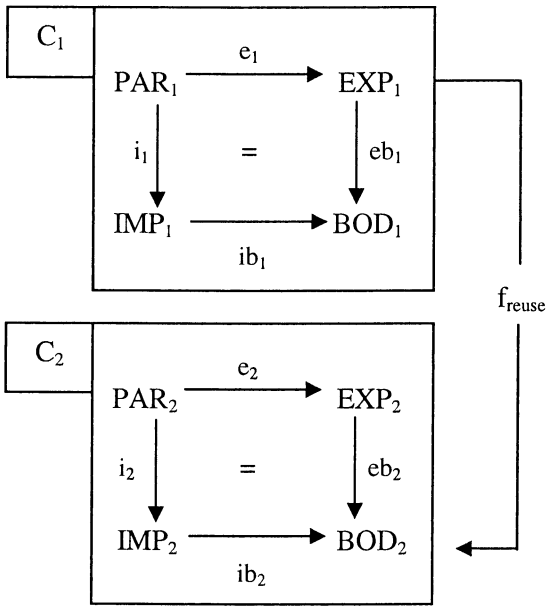
ahol

$$EXP_1=(S_1, OP_1, E_1), EXP_2 = (S_2, OP_2, E_2).$$

A C_2 típusosztály specifikációját a C_1 típusosztály specifikációjából újrafelhasználással történő öröklődésnek nevezzük, ha létezik egy

f_{reuse}: $EXP_1 \rightarrow BOD_2$ kitüntetett elemű specifikációmorfizmus a következő ábrának megfelelően:





Itt az f_{reuse} specifikációmorfizmus lehet

1. Identitás.
2. Újraelnevezés:
   - A szort új nevet kap;

     sort =

     <új név> = <régi név>
   - Az operáció új nevet kap;

     oprs =

     <új opr. név> = <régi opr.név>

3. Tartalmazás:
   - szortok esetében bővül a szortok listája;
   - operációk esetében újabb operációszimbólumokkal bővül a lista;
   - újabb axiómákkal bővül a lista, az axiómákban a változók helyén az új nevű szortok ill. új szortok példányai jelennek meg.

7.2.1. PÉLDA

bag(data, n: **nat**) **is a class specification** =

parameters = nat + bool +

sorts: data

oprs: $\perp$: $\rightarrow$data

=: data data $\rightarrow$ bool [infix]

eqns: a, b, c$\in$data;

a = b $\Rightarrow$ b = a

a = b $\wedge$ b = c $\Rightarrow$ a = c

exports =

class sorts: bag

oprs: bempty: $\rightarrow$ bag

binsert: bag data $\rightarrow$ bag

bdelete: bag data $\rightarrow$ bag

many: bag data $\rightarrow$ nat

bsize: bag $\rightarrow$ nat

eqns: b$\in$bag; a, e$\in$data;

bdelete(bempty, a) = bempty

many(bempty, a) = zerus

bsize(bempty) = zerus

bdelete(binsert(b, a), e) = **if** a = e **then** b

else binsert(bdelete(b, e)) **fi**

many(binsert(b, a), e) = **if** a = e **then** suc(many(b))

else many(b, e) **fi**

bsize(binsert(b, a)) = succ(bsize(b))

bsize(binsert(b, a)) > n ⇒

binsert(s, a) = " undefined"

imports =

sorts: vector

oprs: nil:→vector

put: vector nat data → vector

access: vector nat → data

$shift_L$: vector nat → vector

eqns: v∈ vector; i, j∈ nat; d∈ data;

access(nil, i) =⊥

$shift_L$(nil) = nil

access(put(v, i, d), j)=

if i = j **then** d **else** access(j) **fi**

i =j ⇒ $shift_L$(put(v, i, d), j) = $shift_L$(v, j)

1 < i < j ⇒

$shift_L$(put(v i, d), j) = put($shift_L$(v, j), i, d)

i > j ⇒

$shift_L$(put(v, i, d), j) = put($shift_L$(v, j), prec(i), d)

body =

oprs: rep: vector vector nat → bag

eqns: v∈ vector(nat, data); u∈ vector(nat, nat);

m∈ nat;d∈ data; 0 = zerus; 1 = succ(zerus);

```
                bempty = (nil, nill, 0)
                binsert(v, u, m, d) =
                    if (∃j, 1 ≤ j ≤ m)(access(v, j) = d) then
                    (v, put(u, j, succ(access(u, j)), succ(m)) else
                    (put(v, succ(m), d), put(u, succ(m), 1), succ(m)) fi
end bag;

set(data, n: nat) is a reuse of bag =
    parameters = bool + nat +
        sorts: data
        oprs: ⊥: → data
              =: data data → bool [infix]
        eqns: a, b, c∈data;
              a = b ⇒ b = a
              a = b ∧ b = c ⇒ a = c
    exports =
        class sorts: set
        oprs: empty: → set
              insert: set data → set
              delete: set data → set
              has: set data → bool
              size: set → nat
        eqns: s∈set; a, b∈data;
              delete(empty, a) = empty
              has(empty, a) = "false"
              size(empty) = zerus
```

delete(insert(s, a), b) = **if** a = b **then** delete(s, b)

else insert(delete(s, b), a) **fi**

has(insert(s, a), b) = **if** a = b **then** "true"

else has(s, b) **fi**

size(insert(s, a)) = **if** has(s, a) **then** size(s)

else succ(size(s)) **fi**

size(insert(s, a)) > n ⇒ insert(s, a) = "undefined"

imports =

body =

class sorts: set = bag

eqns: b∈bag, d∈data;

empty = bempty

insert(b, d) = **if** many(b, d) = 0 **then** binsert(d, b)

else b **fi**

delete(b, d) = **if** many(b, d) = 0 **then** b

else bdelete(b, d) **fi**

has(b, d) = many(b, d) > 0

size(b) = bsize(b)

end set;

II. RÉSZ

KONKURENS RENDSZEREK

A KONKURENS PROGRAMOZÁS ALAPFOGALMAI

A PÁRHUZAMOS PROGRAMOZÁS SZEMLÉLTETŐ PÉLDÁI

DISZJUNKT DETERMINISZTIKUS PROGRAMOKBÓL ÁLLÓ PÁRHUZAMOS PROGRAMOK

PÁRHUZAMOS PROGRAMOK KÖZÖS HASZNÁLATÚ VÁLTOZÓKKAL

PÁRHUZAMOS PROGRAMOK SZINKRONIZÁCIÓVAL

A KÖLCSÖNÖS KIZÁRÁS PROBLÉMÁJA

A KIÉHEZTETÉS

EGY MÓDSZER PÁRHUZAMOS PROGRAMOK ELŐÁLLÍTÁSÁRA

OSZTOTT RENDSZEREK

OBJEKTUMOKHOZ VALÓ HOZZÁFÉRÉS SPECIFIKÁCIÓJA A SZINKRONIZÁCIÓ KÖVETELMÉNYEINEK MEGFELELŐEN

A SZINKRONIZÁCIÓS SPECIFIKÁCIÓ ELEMZÉSE

EGY MÓDSZER A SZINKRONIZÁCIÓS SPECIFIKÁCIÓ KONKRÉT MEGADÁSÁRA

A KONKRÉT SPECIFIKÁCIÓ ELEMZÉSE

1. A KONKURENS PROGRAMOZÁS ALAPFOGALMAI

DEFINÍCIÓ A **konkurens program** egy (P, E) kettős, ahol

P: **folyamatok** véges halmaza,

E: **közös használatú erőforrások** véges halmaza.

A **folyamat** végrehajtás alatt lévő szekvenciális program, amely párhuzamosan fut más folyamatokkal együtt azonos programon belül, és ezek a folyamatok feladatuk közös megoldása érdekében egymással **kommunikálnak**.

Közös használatú erőforrás közös használatú objektumot jelent. A mi esetünkben az objektum

- közös használatú memória (közös használatú egyszerű vagy indexes változó a programozási nyelvben);
- közös használatú adattípus objektuma;
- közös használatú kommunikációs csatorna.

A **kommunikáció** azt jelenti, hogy a konkurens folyamatok egyike befolyásolja a másik párhuzamosan futó komponens futását közös változók felhasználásával ill. üzenetküldéssel.

A kommunikáció során az egyik folyamat beállítja a közös változó értékét, ill. üzenetet küld, amelyet a másik folyamat elolvas. Ez a kommunikáció akkor sikeres, ha az érték, ill. üzenet kiolvasásának megtörténtéről a címzett folyamat

a kommunikációs partnerének egyértelmű választ ad, még mielőtt az érték, ill. az üzenet megsemmisülne. A sikeres kommunikáció neve **szinkronizáció**.

Közös használatú memória esetén a memória celláinak szinkronizáltaknak kell lenniük olvasásra és írásra. Ez a programozás szintjén cellák helyett közös használatú változókra vonatkozó állítást jelent.

Kommunikációs hálózat esetén, ha a P_2 folyamat küld egy üzenetet a C csatornán keresztül a P_1 folyamatnak és az üzenet megérkezett P_1-hez, akkor annak erre általában vissza kell jeleznie. Ez P_1 és környezetének bizonyos szinkronizált voltát kívánja meg.

OSZTOTT RENDSZER Az osztott rendszer egymástól elkülönülten létező folyamatkomponensek egy halmaza. Ezek a komponensek egymástól függetlenül dolgoznak felhasználva saját erőforrásukat. Azonban időről időre üzeneteket küldenek egymásnak.

A kommunikációs hálózatok ismert példái a web-alkalmazások architektúrái. A web-alkalmazások a tradicionális funkcionális alkalmazások osztott rendszer felhasználásával történő megoldásának egy formáját jelentik.

A web-alkalmazások architekturái többszintű architektúrák. Általában van egy úgynevezett kliens feladatokat ellátó "front-end" komponens, amely rendszerint a web-böngészőt is tartalmazza, egy szerver feladatokat ellátó "back-end" komponens, amely tartalmazza az adaterőforrásokat, és lehetnek még közbülső szintek is, amelyek bizonyos web-szolgáltatást és alkalmazási funkciókat nyújtanak.

A web-alkalmazások meglehetősen egyszerű szerkezetűek:

- szüksége van egy böngésző programra a felhasználónak a saját gépén.
- kell egy web-szerver.

- kell egy bizonyos nyelv, például a HTML.

Amikor a felhasználó rákattintással kiválasztja a szolgáltatást, a böngésző összeköttetést létesít közte és a szerver között, valamint kéri a lapot. A szerver elemzi a kérést, kikeresi a kért lapot és átadja azt a böngészőnek. Amikor a kérés teljesült, a kapcsolat megszűnik.

A rendszer akkor válik bonyolulttá, ha újabb rétegek lépnek be a tevékenységbe, például adatbázis lekérdezése, felújítása stb.

AZ ÜZENETFORMÁBAN TÖRTÉNŐ KOMMUNIKÁCIÓ SZERVEZÉSI FORMÁI Az üzenetformában történő kommunikációnak a következő típusai alakultak ki a gyakorlat során:

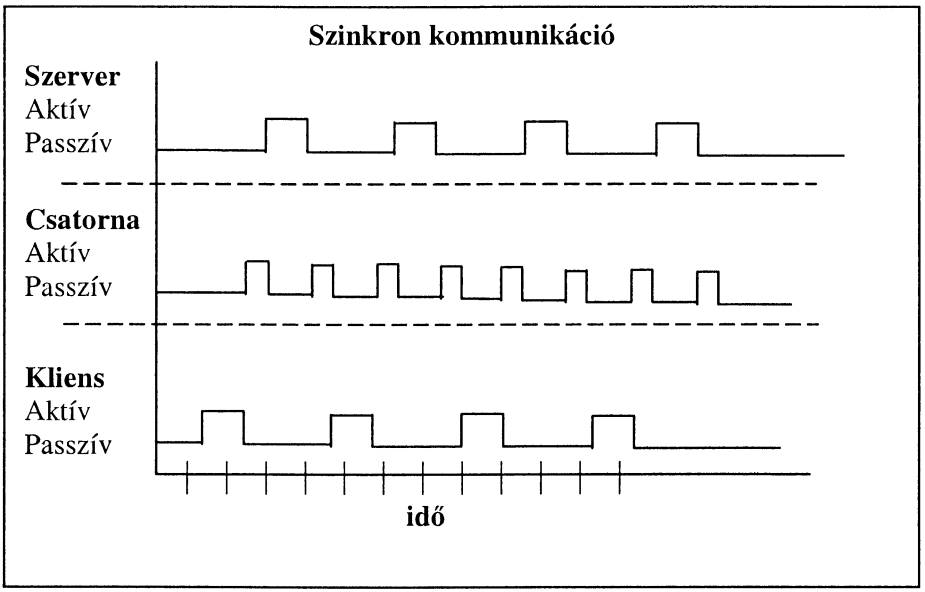
- szinkron kommunikáció.
- aszinkron kommunikáció.
- késleltetett szinkron kommunikáció.
- visszahívásos aszinkron kommunikáció.

A kommunikáció csatornán keresztül történik. A csatorna tulajdonságai általában a következők:

- A csatorna nem irányított csatorna, azaz mindkét irányban alkalmas arra, hogy adatokat továbbítson. Más szóval ez azt jelenti, hogy mind input, mind output utasítások értelmezve vannak rajta.
- A csatornának nincs típusa, azaz bármilyen típusú adat továbbítására alkalmas.

Ezek után tekintsük át a kommunikáció fenti formáit egy-egy szemléltető ábra felhasználásával.

SZINKRON KOMMUNIKÁCIÓ CSATORNÁN KERESZTÜL A küldő az üzenetet akkor tudja átadni, ha a fogadó az adott időpontban kész annak átvételére. (telefonálás). Ezt szemlélteti a következő ábra.



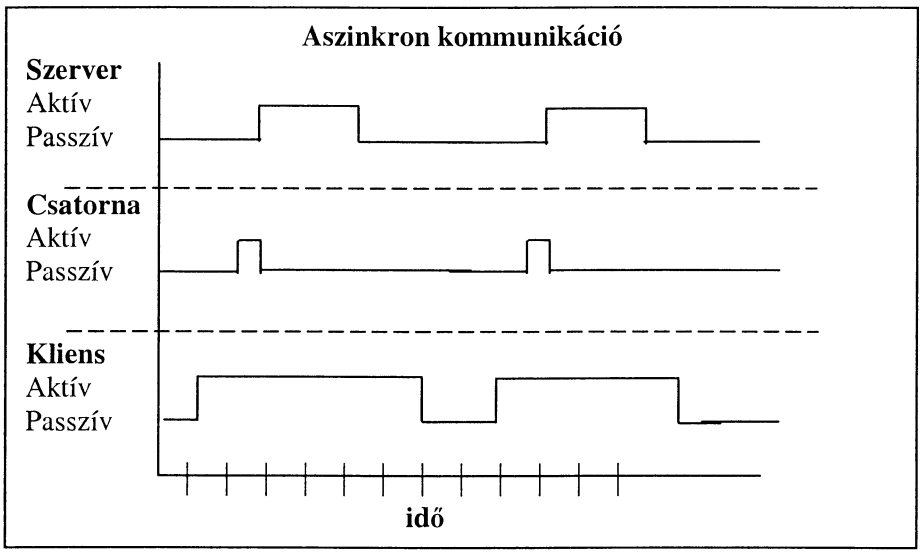
Aktív állapot: az objektum a csatornán történő kommunikáció végrehajtásának lebonyolítása érdekében tevékenykedik.

Passzív álapot: az objektum nem foglalkozik a csatornán történő kommunikáció lebonyolításával.

Kezdeményező: a kliens.

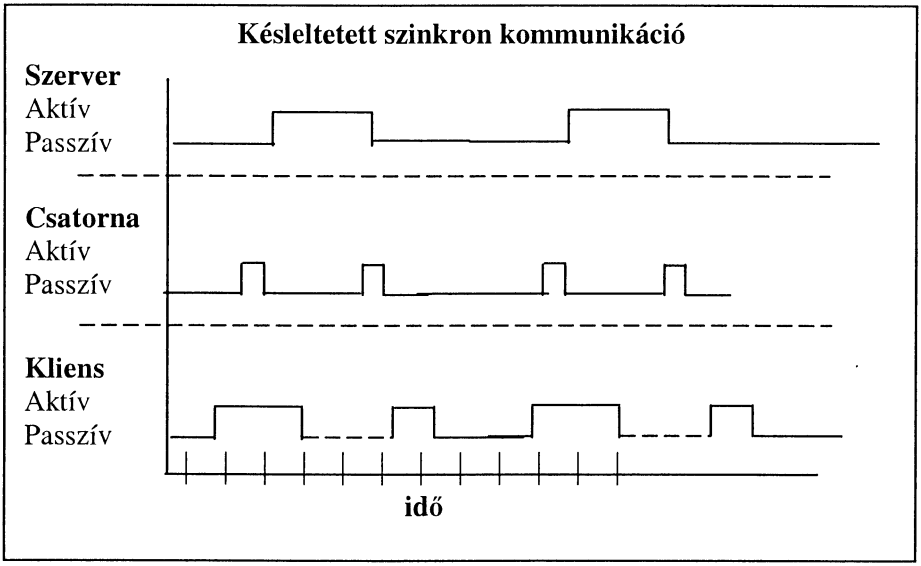
Miután a kliens kiadta a csatornára vonatkozó utasítást, az a csatornán keresztül eljut a szerverhez, amely teljesíti a kérést, és a választ a csatornán keresztül eljuttatja a klienshez. A kliens csak ekkor kezd hozzá a válasz feldolgozásához. A csatorna használatának ez a folyamata ismétlődhet egymás után.

ASZINKRON KOMMUNIKÁCIÓ CSATORNÁN KERESZTÜL A küldő az üzenetet akkor is el tudja küldeni, ha a fogadó éppen nem kész annak az átvételére (levélküldés). A következő ábra ezt a folyamatot szemlélteti.



A kliens kezdeményezi a kommunikációt, majd folytatja tevékenységét. A kérés továbbításának és feldolgozásának folyamata a csatornaobjektum és a szerverobjektum esetében azonos az előbbivel, kévéve a válaszadást.

KÉSLELTETETT SZINKRON KOMMUNIKÁCIÓ CSATORNÁN KERESZTÜL A kliens kezdeményezi a kommunikációt, majd folytatja tevékenységét. A kérés teljesítésének folyamata a csatorna és a szerver között ugyanaz, mint a szinkron kommunikáció esetében. Egy idő múlva a kliensnek szüksége lehet a válaszra. Ekkor várakoznia kell arra, hogy megkapja a választ a csatornán keresztül. Ezt a várakozást szemlélteti a szaggatott vonal a következő ábrán.



VISSZAHÍVÁSOS ASZINKRON KOMMUNIKÁCIÓ CSATORNÁN KERESZTÜL A kliens kezdeményezi a hívást egy aszinkron csatornautasítással, ezután folytatja a tevékenységét. A csatornautasítás ekkor tartalmazza a visszahívás kérését is. Amikor a szerver feldolgozta a kérést, visszahívja a klienst egy aszinkron utasítással.

A konkurens rendszer létrehozása általában nehéz programozási feladatot jelent. A következőkben egy, a gyakorlatban gyakran előforduló feladat leegyszerűsített modelljével szemléltetjük ezeket a programozási nehézségeket, és rámutatunk a párhuzamos programok szekvenciális programoktól különböző speciális tulajdonságaira. Ez a példa hasonló célt szolgál a [KRZ97] munkában is, amelyet mi itt belehelyezünk egy fontos gyakorlati feladat témakörébe és kissé általánosabb formában fogalmazzuk meg azt. Ezután Peterson [BUR93] algoritmusának példáját használjuk fel arra, hogy felhívjuk az olvasó figyelmét az atomi műveletek megválasztásának nagy körültekintést igénylő problémájára.

2. A PÁRHUZAMOS PROGRAMOZÁS PROBLÉMAKÖRÉNEK SZEMLÉLTETŐ PÉLDÁI

A következőkben először a [KRZ97] cikkben szereplő példa egy általánosabb formáját használjuk fel a párhuzamos programozás problémáinak szemléltetésére, rámutatva ennek az egyszerű problémának a gyakorlati jelentőségére.

Gyakran felmerülő probléma, hogy keressük azokat az objektumokat, amelyek az általunk megkívánt tulajdonsággal rendelkeznek. Például keressük a bűnüldöző szervek nyilvántartásában azokat a személyeket, akik egy új bűntény végrehajtásához hasonló tulajdonságokkal rendelkező bűntényt már korábban elkövettek. Ebben az esetben a bűntény körülményei, jellemzői a bemenő adatok, és a keresett személy vagy személyek azonosítói a keresett eredményadatok. Ilyenkor rendszerint terjedelmes adatbázisról van szó, – például nemzetközi bűnözők is szóba jöhetnek, – és a keresés ezért hosszú időt vehet igénybe. Egy lehetséges megoldás az, hogy magát az adatbázist megfelelő módon részekre osztjuk, például országonként, és az egyes részadatbázisokban egymással párhuzamosan dolgozó programok keresik a probléma megoldását. Ennek egy leegyszerűsített modellje lehet a következő.

A keresett objektum tulajdonsága egy adattal jellemezhető. Az objektum azonosítója egy egydimenziós vektor indexe. A részadatbázisokat tehát ebben az esetben felfoghatjuk úgy, hogy azok egydimenziós végtelen vektorok. Speciális

esetként feltehetjük, hogy csak egy olyan részvektor van, amely a keresett adatot tartalmazza. Ebben az esetben tehát, ha valamelyik párhuzamosan futó program megtalálta a megadott adathoz tartozó indexet, akkor ezt jelzi a vele párhuzamosan futó programoknak is, és erre azok befejezik a keresést. Fogalmazzuk meg tehát a feladatot erre a speciális esetre.

2.1. PÉLDA Legyenek $X_1, \ldots, X_n$ végtelen indexű vektorok. Tegyük fel, hogy valamelyikben van olyan index, amelyhez tartózó érték a bemenő adatként megadott e érték. Készítsünk párhuzamos programot annak a j indexnek a megkeresésére, amelyre $X_i[j] = e$, $1 \leq j \leq n$.

A feladat megoldására tehát egy olyan párhuzamos programot kell készíteni, ahol n párhuzamosan futó komponens keresi az e értékhez tartozó indexet, mindegyik a saját X_i vektorában. Ha közülük az egyik megtalálta a keresett indexet, a program befejezi a feladatát.
A feladat megoldására különböző megközelítéseket mutatunk be azzal a céllal, hogy szemléltessük a párhuzamos programozásban jelentkező nehézségeket és problémákat.

Adott tehát a részadatbázisok következő típusspecifikációja:

adatbázis(elem) **is a type specification** =
parameters = **nat** + **bool** +
sorts: elem
oprs: =: elem elem $\rightarrow$ bool [infix]
eqns: a, b, c$\in$elem;
$a = b \Rightarrow b = a$
$a = b \wedge b = c \Rightarrow a = c$
exports =

class sorts: adatbázis

oprs: new: → adatbázis

put: adatbázis nat elem → adatbázis

_ [_] = _ : adatbázis nat elem → bool

eqns: a∈adatbázis; i, j∈nat; e∈elem;

new[j] = e = "false"

put(a, j, e)[i] =e = **if** j = i **then** "true"

else a[i] = e **fi;**

end adatbázis;

Minden X_j, $j \in \{1, \ldots, n\}$, tehát ebbe az adattípusba tartozik.

Feltevésünk az, hogy $(\exists i)(i \in \{1, \ldots, n\} \wedge j \geq 0)(X_i[j] = e)$

A megoldás vázlata a következő lehet:

S: k ← e; **parbegin** $S_1 \parallel \ldots \parallel S_n$ **parend**;

ahol az S_j (j = 1,..., n) párhuzamosan futó komponensek rendre az X_j (j = 1, ..., n) részadatbázisokban keresik a megadott értékhez tartozó indexet.

A megoldás első megközelítése a párhuzamos komponensekre:

...

S_j:

B ← "false"; j ← 0;

while B = "false" **do**

B ← X[j] = k;

j ← j + 1; **od**;

A problémák szemléltetéséhez célszerű a továbbiakban azzal a speciális esettel foglalkozni, ahol n = 2. Mivel két vektorban való keresést vizsgálunk együtt, jelöljük ezeket egyszerűen a következőképpen: X és Y.

Az elkészítendő program specifikációja így a következő:

$\{(\exists l, l \in \text{index})(X[l] = e \vee Y[l] = e)\}$ S $\{(X[j] = e \vee Y[i] = e)\}$

1. A MEGOLDÁS ELSŐ KÍSÉRLETE

S: $k \leftarrow e$;

parbegin $S_1 \parallel S_2$ **parend**;

$z =$ **if** $X[j] = e$ **then** j **else** i **fi**;

S_1: V_0:	S_2: U_0:
V_1: $B \leftarrow$ "false"; $j \leftarrow 0$;	U_1: $B \leftarrow$ "false"; $i \leftarrow 0$;
V_2: **while** $B =$ "false" **do**	U_2: **while** $B =$ "false" **do**
V_3: $B \leftarrow X[j] = k$;	U_3: $B \leftarrow Y[i] = k$;
V_4: $j \leftarrow j + 1$; **od**;	U_4: $i \leftarrow i + 1$; **od**;
V_5:	U_5:

Kérdés, hogy helyes megoldás-e ez.

Tegyük fel, hogy pontosan egy olyan "a" indexérték létezik, amelyre

$$X[a] = e \text{ és } (\sim\exists i)(Y[i] = e).$$

Tekintsük a programnak azt a következő végrehajtását, amikor az első komponens eljutott az "a" indexértékig, miközben a második komponens még nem indult el. A programnak ezt a végrehajtását szemlélteti lépésenként a következő táblázat.

S_1	S_2	B	j	i	
V_2	U_0	"false"	a	0	
V_3	U_0	"false"	a	0	$B \leftarrow (X[a] = e)$;
V_4	U_1	"true"	a	0	$B \leftarrow$ "false"; $i \leftarrow 0$; $a \leftarrow a + 1$;
V_2	U_2	"false"	a + 1	0	
Végtelen ciklus!					

Itt tehát S_1, miután megtalálta a keresett indexértéket, B értékét igaz értékre beállította. Ekkor azonban elindult S_2 és visszaállította a B értékét hamis értékre, még mielőtt S_1 kiértékelhette volna a ciklusfeltételt. Így előáll a végtelen ciklus.

Mivel mindkét komponensben kétszer adunk értéket a B változónak, kézenfekvő ennek megszüntetése révén keresni a jó megoldást.

2. A MEGOLDÁS MÁSODIK KÍSÉRLETE

S: k ← e; B ← "false";

parbegin S_1 || S_2 **parend**;

z = **if** X[j] = e **then** j **else** i **fi**;

S_1:	S_2:
V_0: j ← 0;	U_0: i ← 0;
V_1: **while** B = "false" **do**	U_1: **while** B = "false" **do**
V_2: B ← X[j] = k;	U_2: B ← Y[i] = k;
V_3: j ← j + 1; **od**;	U_3: i ← i + 1; **od**;
V_4:	U_4:

Vizsgáljuk most ismét az előbbi esetet.

Feltevés: pontosan egy olyan "a" indexérték létezik, amelyre

$$X[a] = e \text{ és } (\sim\exists i)(Y[i] = e).$$

Tegyük fel ismét, hogy az első komponens előreszaladt és megtalálta ezt az értéket. Ekkor lép be a második komponens egy újabb iterációba a következők szerint:

S_1	S_2	B	j	i	
V_2	U_1	"false"	a	b	B ← (X[a] = e);
V_3	U_1	"true"	a	b	a ← a + 1;
V_1	U_1	"true"	a + 1	b	
V_4	U_2	"true"	a + 1	b	B ← (X[b] = e);
V_4	U_3	"false"	a + 1	b	b ← b + 1;
V_4	U_1	"false"	a + 1	b + 1	
V_4	U_2	"false"	a + 1	b + 1	
Végtelen ciklus!					

A probléma gyökere tehát az lehet, hogy a párhuzamos komponensen belüli minden egyes iteráció során felülírjuk a B értékét. Ezt szüntetjük meg a következő közelítő megoldási kísérletünk során. Csak akkor adunk új értéket a B változónak, ha megtaláltuk a keresett indexértéket.

3. A MEGOLDÁS HARMADIK KÍSÉRLETE

S′: k ← e; B ← "false"; **parbegin** S_1 || S_2 **parend**; ahol

S_1: V_0: j ← 0;	S_2: U_0: i ← 0;
V_1: **while** B = "false" **do**	U_1: **while** B = "false" **do**
V_2: **if** X[j] = k	U_2: **if** Y[i] = k
then B ← "true" **fi**;	**then** B ← "true" **fi**;
V_3: j ← j + 1; **od**;	U_3: i ← i + 1; **od**;
V_4:	U_4:

Ismét éljünk a korábbi feltevésünkkel:

Feltevés: pontosan egy olyan "a" indexérték létezik, amelyre

$$X[a] = e \text{ és } (\sim\exists i)(Y[i] = e).$$

Legyen ismét az első komponens kellően gyors, amely miután megtalálta a keresett értéket, befejezi a futását. A végrehajtás tehát a következő:

S_1	S_2	B	j	i	
V_2	U_0	"false"	a	0	B ← "true";
V_3	U_0	"true"	a	0	a ← a+ 1;
V_1	U_0	"true"	a + 1	0	
V_4	U_0	"true"	a + 1	0	
Végtelen várakozás!					

Itt azonban, ha a második komponens nem indul el, a párhuzamos program nem fejeződik be. Ez a komponensek beütemezésének kérdése. Az ilyen beütemezést (végrehajtást) nem tisztességes (nem fair) végrehajtásnak nevezzük.

A probléma helyes megoldása tehát az lehet, ha megköveteljük, hogy minden komponens sebessége pozitív érték legyen. De a párhuzamos komponenseknek egymással szinkronban kell dolgozniuk. Ezért azt, hogy a komponensek sebessége mindig pozitív legyen, nem követelhetjük meg, mert meg kell engedni bizonyos események bekövetkezésére történő várakozásokat is. Ehhez vezessük be a következő megvárakoztató utasítást:

await B **then** S **ta**;

Ennek jelentése az, hogy az utasítást kiadó folyamat álljon meg, ha a kiértékelésnél B értéke hamis, és folytassa a futását, ha igaz, de úgy, hogy az S programot meg nem szakítható, atomi utasításként hajtsa végre.

Használjuk tehát a következőkben ezt az utasítást, a komponensek pedig egymást váltogatva végezzék a keresést.

4. A MEGOLDÁS NEGYEDIK KÍSÉRLETE

S'''**:** k ← e; next ← 1; B ← "false"; **parbegin** S_1 || S_2 **parend**;

z = **if** X[j] = e **then** j **else** i **fi**;

S_1: j ← 0;	S_2: i ← 0;
V_1: **while** B = "false" **do**	U_1: **while** B = "false" **do**
V_2: **await** next = 1	U_2: **await** next = 2
then next ← 2 **ta**;	**then** next ← 1 **ta**;
V_3: **if** X[j] = k	U_3: **if** X[i] = k
then B ← "true" **fi**;	**then** B ← "true" **fi**;
V_4: j ← j + 1; **od**;	U_4: i ← i + 1; **od**;
V_5:	U_5:

Ismét legyen a feltevésünk a korábbi:

Feltevés: pontosan egy olyan "a" indexérték létezik, amelyre

$$X[a] = e \text{ és } (\sim\exists i)(Y[i] = e).$$

Tekintsük a program következő végrehajtását:

S_1	S_2	B	next	j	i
V_3	U2	"false"	2	a	b
V_3	U3	"false"	1	a	b
V_3	U4	"false"	1	a	b + 1
V_3	U1	"false"	1	a	b + 1
V_4	U2	"true"	1	a + 1	b + 1
V_1	U2	"true"	1	a + 1	b + 1
V_5	U2	"true"	1	a + 1	b + 1
Holtpontállapot (deadlock).					

Itt az első komponens megtalálta a keresett értéket, miközben a második komponens ugrásra készen várakozott az U_2 címkénél. Miután az első komponens megadta számára a jelet, hogy folytathatja futását, az első komponens a V_3 címkénél lévő utasítás végrehajtásával bajlódott, miközben az első komponens eljutott ismét az U_2 címkéhez. Ekkorra azonban S_1 már a B értékét igaz értékre állította be. Az első komponens kiléphetett a ciklusból, futását befejezte anélkül, hogy a next értékét felülírta volna. Így a második komponens végtelen várakozásra kényszerült az U_2 címkénél. Ezt nevezzük holtpontállapotnak.

A holtpont elkerülésének egyik útja lehet, ha mindkét komponens akkor is gondoskodik a másik komponens továbbengedéséről, ha kilépett a ciklusból. Ilyen megoldás lehet a következő.

5. A MEGOLDÁS ÖTÖDIK KÍSÉRLETE

S: k ← e; next ← 1; B ← "false"; **parbegin** S_1 || S_2 **parend**;

S_1: j ← 0;	S_2: i ← 0;
while B = "false" **do**	**while** B = "false" **do**
await next = 1	**await** next = 2
then next ← 2 **ta**;	**then** next ← 1 **ta**;
if X[j] = k	**if** Y[i] = k
then B ← "true" **fi**;	**then** B ← "true" **fi**;
j ← j + 1; **od**; next ← 2;	i ← i - 1; **od**; next ← 1;

Ez tehát már a probléma helyes megoldását nyújtja, megengedve azt, hogy a folyamatok sebessége utasítás hatására zérus is lehet.

Példánk alapján összefoglalhatjuk, hogy a párhuzamos programok helyességének kimutatása során a következő tulajdonságok képezik a vizsgálat tárgyát:

1. Parciális helyesség.
2. Befejeződés.
3. Teljes helyesség.
4. Hibamentesség.
5. Interferenciamentesség.
6. Holtpontmentesség.

2.2. PÉLDA A kölcsönös kizárás Peterson-féle algoritmusa [PET83]. Ezzel a példával azt szemléltetjük, hogy párhuzamos programokban rendkívül fontos annak eldöntése, hogy mi az a művelet, ami meg nem szakítható (atomi) műveletként fog a program végrehatása során viselkedni.

Készítsünk olyan programot, amelyben két végtelen ciklusban párhuzamosan futó folyamat közül mindig csak legfeljebb az egyik folyamat tartózkodhat a kritikus szakaszában. Ezt nevezzük kölcsönös kizárásnak.

1. MEGOLDÁS

P: $s \leftarrow 1$; $y_1 \leftarrow 0$; $y_2 \leftarrow 0$; **parbegin** $S_1 \parallel S_2$ **parend**;

S_1: **while** "true" **do**	S_2: **while** "true" **do**
nem kritikus szakasz;	nem kritikus szakasz;
$(y_1, s) \leftarrow (1, 1)$;	$(y_2, s) \leftarrow (1, 2)$;
wait $(y_2 = 0) \vee (s \neq 1)$;	**wait** $(y_1 = 0) \vee (s \neq 2)$;
kritikus szakasz$_1$;	kritikus szakasz$_2$;
$y_1 \leftarrow 0$; **od**;	$y_2 \leftarrow 0$; **od**;

A **wait** b utasítás mindkét párhuzamosan futó folyamat esetében a forgalmi lámpa szerepét tölti be, a folyamat akkor futhat tovább, ha a logikai kifejezés értéke igaz értéket ad. Itt most a két változónak egyszerre történő értékadást atomi utasításként értelmezzük Ezzel a feltételezéssel könnyen belátható, hogy ez a program megvalósítja a kölcsönös kizárást a kritikus szakaszok között.

2. A PETERSON-FÉLE ALGORITMUS MÁSODIK VÁLTOZATA

Mi történik akkor, ha a $(y_1, s) \leftarrow (1, 1)$ vektorművelet nem hajtható végre atomi utasításként? Vizsgáljuk meg ennek a következményeit! Kapjon először s értéket, majd utána y_1 ill. y_2.

$s \leftarrow 1$; $y_1 \leftarrow 0$; $y_2 \leftarrow 0$; **parbegin** $S_1 \| S_2$ **parend**;

S_1:	S_2:
U_0: **while** "true" **do**	V_0: **while** "true" **do**
U_1: nem kritikus szakasz$_1$;	V_1: nem kritikus szakasz$_2$;
U_2: $s \leftarrow 1$;	V_2: $s \leftarrow 2$;
U_3: $y_1 \leftarrow 1$;	V_3: $y_2 \leftarrow 1$;
U_4: **wait** $(y_2 = 0) \vee (s \neq 1)$;	V_4: **wait** $(y_1 = 0) \vee (s \neq 2)$;
U_5: kritikus szakasz$_1$;	V_5: kritikus szakasz$_2$;
U_6: $y_1 \leftarrow 0$; **od**;	V_6: $y_2 \leftarrow 0$; **od**;

Vizsgáljuk most ennek a megoldásnak a következő táblázat szerinti végrehajtását.

S_1	S_2	s	y_1	y_2
U_0	V_0	1	0	0
U_1	V_0	1	0	0
U_1	V_1	1	0	0
U_2	V_1	1	0	0
U_2	V_2	1	0	0
U_3	V_2	1	0	0
U_3	V_3	2	0	0
U_3	V_4	2	0	1
U_3	V_5	2	0	1
U_4	V_5	2	1	1
U_5	V_5	2	1	1

Mind az S_1 program, mind az S_2 program a kritikus szakaszba jutott el. Nem teljesül a kölcsönös kizárás. Ezt a hibát tehát a korábbi atomi művelet két olyan műveletre való bontása okozta, amelyben a vektor komponensei egyenként atomi utasításként kapnak értéket. De lássunk csodát! Cseréljük csak meg ennek a két értékadásnak a sorrendjét!

3. A PETERSON-FÉLE ALGORITMUS HARMADIK VÁLTOZATA

$s \leftarrow 1$; $y_1 \leftarrow 0$; $y_2 \leftarrow 0$; **parbegin** $S_1 \| S_2$ **parend**;

S_1:	S_2:
U_0: **while** "true" **do**	V_0: **while** "true" **do**
U_1: nem kritikus szakasz$_1$;	V_1: nem kritikus szakasz$_2$;
U_2: $y_1 \leftarrow 1$;	V_2: $y_2 \leftarrow 1$;
U_3: $s \leftarrow 1$;	V_3: $s \leftarrow 2$;
U_4: **wait** $(y_2 = 0) \vee (s \neq 1)$;	V_4: **wait** $(y_1 = 0) \vee (s \neq 2)$;
U_5: kritikus szakasz$_1$;	V_5: kritikus szakasz$_2$;
U_6: $y_1 \leftarrow 0$; **od**;	V_6: $y_2 \leftarrow 0$; **od**;

Könnyen beláthatjuk, hogy ez a program már korrekt implementáció. Formálisan is bebizonyítható, hogy esetében teljesül a kölcsönös kizárás. A formális bizonyításra később vissza fogunk térni.

3. DISZJUNKT DETERMINISZTIKUS PROGRAMOKBÓL ÁLLÓ PÁRHUZAMOS PROGRAMOK

A DISZJUNKT PÁRHUZAMOS PROGRAM INFORMÁLIS DEFINÍCIÓJA A párhuzamos program determinisztikus komponenseinek folyamatai közös feladatot oldanak meg, de nincs olyan közös erőforrásuk, amelynek a belső állapotát megváltoztatnák.

Legyen S determinisztikus program, és vezessük be a következő jelöléseket:

var(S): az S program összes változóinak halmaza.

change(S): az S program azon változóinak halmaza, amelyek az S programban egy értékadás bal oldalán megjelennek, amelyeknek tehát az értékét, azaz az állapotát az S program megváltoztatja.

Diszjunkt determinisztikus párhuzamos program szintaxisa

$S ::=$ **skip** $\mid y \leftarrow f(x, y) \mid S_1; S_2 \mid$ **if** $\alpha(x, y)$ **then** S_1 **else** S_2 **fi** $\mid$
while $\alpha(x, y)$ **do** S_1 **od**; $\mid$
parbegin $S_1 \parallel S_2$ **parend**;

ahol $var(S_1) \cap change(S_2) = \{\} \wedge var(S_2) \cap change(S_1) = \{\}$

3.1. PÉLDA diszjunkt programok var(S) és change(S) függvényeinek felírására:

S_1: $i \leftarrow 1$; $x \leftarrow 1$; **while** $i \leq m$ **do** $x \leftarrow x \times a$; $i \leftarrow i + 1$ **od**;

$$S_2: y \leftarrow 2; z \leftarrow (a + 1) \times (a - 1);$$

$$var(S_1) = \{i, x, m, a\}, change(S_1) = \{i, x\};$$

$$var(S_2) = \{y, z, a\}, change(S_2) = \{y, z\}.$$

Nemcsak a diszjunkt programok fogalmára lesz szükségünk, hanem programok és kifejezések diszjunkt voltára is.

DEFINÍCIÓ Jelölje var(E) az E kifejezésben szereplő változók halmazát. **Az S program és az E kifejezés diszjunkt**, ha

$$change(S) \cap var(E) = \{\}.$$

Például $y \leftarrow z + (3 * x + 1)$; S: $y \leftarrow x$; esetén a $z + (3 * x + 1)$ kifejezés és az S program diszjunktak, mivel

$$change(y \leftarrow x) = \{y\},$$

$$var\ (z + (3 * x + 1)) = \{z, x\}.$$

DEFINÍCIÓ Az S: **parbegin** $S_1 \parallel \ldots \parallel S_n$ **parend;** diszjunkt párhuzamos program végrehajtása akkor és csak akkor fejeződik be, ha minden S_i $(1 \leq i \leq n)$, végrehajtása befejeződik.

AXIÓMA A párhuzamos programszerkezethez tartozó állapotátmenet:

$$\frac{\langle S_i, \sigma\rangle \rightarrow \langle T_i, \sigma'\rangle,\ i \in \{1, \ldots, n\}}{\langle \textbf{parbegin}\ S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_n\ \textbf{parend}, \sigma\rangle \rightarrow \langle \textbf{parbegin}\ S_1 \parallel \ldots \parallel T_i \parallel \ldots \parallel S_n\ \textbf{parend}, \sigma'\rangle}$$

DEFINÍCIÓ A program két kiszámítását akkor nevezzük I/O-ekvivalensnek, ha azonos állapotból kiindulva azonos eredményállapotot eredményeznek, akár véges, akár végtelen kiszámítási sorozatról van szó.

3.1. ÁLLÍTÁS Legyenek S_1, ..., S_n páronként diszjunkt determinisztikus programok. Akkor

$$M[\textbf{parbegin } S_1 \parallel \ldots \parallel S_n \textbf{ parend}] = M[S_1; \ldots; S_n]$$

és

$$M_{tot}[\textbf{parbegin } S_1 \parallel \ldots \parallel S_n \textbf{ parend}] = M_{tot}[S_1; \ldots; S_n].$$

BIZONYÍTÁS Mivel S_i, S_j $i \neq j$ esetén diszjunktak, ezért ha x az S_i egy változója, akkor annak értékét S_j végrehajtása nem változtatja meg. Így $\langle S_i; S_j, \sigma\rangle$ minden σ mellett helyettesíthető az $\langle S_j; S_i, \sigma\rangle$ konfigurációval. Tehát

$$M(S_i; S_j)(\sigma) \subseteq M(S_j; S_i)(\sigma).$$

Ezért

$$M[\textbf{parbegin } S_i \parallel S_j \textbf{ parend}] = M[S_i; S_j].$$

Ebből indukcióval, figyelembe véve az I/O-ekvivalenciát, azonnal adódik, hogy

$$M[\textbf{parbegin } S_1 \parallel \ldots \parallel S_n \textbf{ parend}] = M[S_1; \ldots; S_n],$$

Hasonlóképpen

$$M_{tot}[\textbf{parbegin } S_1 \parallel \ldots \parallel S_n \textbf{ parend}] = M_{tot}[S_1, \ldots; S_n].$$

DISZJUNKT PROGRAMOK SZEKVENCIALIZÁCIÓS SZABÁLYA:

$$\frac{\{p\}\ S_1; \ldots; S_n\ \{q\}}{\{p\}\ \textbf{parbegin } S_1 \parallel \ldots \parallel S_n \textbf{ parend}\ \{q\}}$$

3.2. PÉLDA Bizonyítsuk be a következő tételt:

$$\{x < y \wedge (0 < a \leq b)\}\ \textbf{parbegin } x \leftarrow a * x \parallel y \leftarrow b * y \textbf{ parend}\ \{x < y\}$$

BIZONYÍTÁS Ez egy diszjunkt párhuzamos program, ahol az

$$x \leftarrow a * x \; ; \; y \leftarrow b * y$$

determinisztikus programra igaz az

$$\{x < y \wedge (0 < a \leq b)\} \; x \leftarrow a * x; \; y \leftarrow b * y \; \{x < y\}$$

tétel. Ezért a szekvencializációs szabály alapján a tétel állítása igaz.

Jelölés: free(p): a p predikátumban az összes olyan változó halmaza, amely szabadon fordul elő a p predikátumban, azaz amelyre az állításban kvantor nem vonatkozik. Például legyen

$$p: x > 0 \wedge (\exists y, y > 0)(x = y * y).$$

Itt x szabadon fordul elő, y viszont nem. Tehát most free(p) = {x}.

A szekvencializációs szabály általánosítása a **diszjunkt párhuzamossági szabály**:

$$\{p_i\} \; S_i \; \{q_i\}, \; 1 \leq i \leq n$$

és

$$(\forall i, j, i \neq j) \Rightarrow (\text{free}(p_i) \cap \text{change}(S_j) = \{\} \wedge \text{free}(q_i) \cap \text{change}(S_j) = \{\})$$

$$\{p_1 \wedge \ldots \wedge p_n\} \; \textbf{parbegin} \; S_1 \parallel \ldots \parallel S_n \; \textbf{parend} \; \{q_1 \wedge \ldots \wedge q_n\}$$

Jelölés:

$$(\forall i, j, i \neq j) \Rightarrow [(\text{free}(p_i) \cap \text{change}(S_j) = \{\} \wedge \text{free}(q_i) \cap \text{change}(S_j) = \{\}) \equiv$$
$$(\text{free}(p_i, q_i) \cap \text{change}(S_j) = \{\})].$$

A szabály általánosabb formája:

$$p \Rightarrow p_1, \ldots, p \Rightarrow p_n;\ \{p_i\}\ S_i\ \{q_i\},\ 1 \le i \le n;\ q_1 \wedge \ldots \wedge q_n \Rightarrow q$$

és

$$(\forall i, j, i \neq j) \Rightarrow (\mathrm{free}(p_i, q_i) \cap \mathrm{change}(S_j) = \{\})$$

$$\{p\}\ \textbf{parbegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{parend}\ \{q\}$$

3.3. PÉLDA Bizonyítsuk be a következő tételt:

$$\{x = y \wedge 0 < a\}\ \textbf{parbegin}\ x \leftarrow a * x \parallel y \leftarrow a * y\ \textbf{parend}\ \{x = y\}$$

BIZONYÍTÁS A szekvencializációs szabály alapján bizonyítandó, hogy

$$\{x = y \wedge 0 < a\}\ x \leftarrow a * x;\ y \leftarrow a * y\ \{x = y\},$$

ami az értékadás következtetési szabálya alapján nyilvánvaló:

$$\{x = y \wedge 0 < a\}\ x \leftarrow a * x\ \{x = a * y\};\ \{x = a * y\}\ y \leftarrow a * y\ \{x = y\}.$$

A diszjunkt párhuzamosság szabályával azonban nem bizonyítható, mivel

$$\mathrm{free}(x = y \wedge 0 < a, x = a * y) \cap \mathrm{change}(y \leftarrow a * y) \neq \{\} \wedge$$

$$\mathrm{free}(x = a * y, x = y) \cap \mathrm{change}(y \leftarrow a * y) \neq \{\};$$

Segédváltozó bevezetésével azonban sikerrel alkalmazhatjuk a diszjunkt párhuzamossági szabályt:

$$\{x = z \wedge y = x \wedge 0 < a\}$$

parbegin

$$\{x = z \wedge 0 < a\}\ x \leftarrow a * z\ \{x = a * z\} \parallel \{y = z \wedge 0 < a\}\ y \leftarrow a * z\ \{y = a * z\}$$

parend

$$\{x = a * z \wedge y = a * z\}$$

Most a

$$(\mathrm{free}(x = z \wedge 0 < a, x = a * z) \cap \mathrm{change}(y \leftarrow a * z) = \{\}) \wedge$$

$$(\mathrm{free}(y = z \wedge 0 < a, y = a * z) \cap \mathrm{change}(x \leftarrow a * z) = \{\})$$

feltétel teljesül.

Baj azonban az, hogy $x = y \Rightarrow x = z \wedge y = z =$ "false".

Megoldás lehet a következő:

$$\{x = y \wedge 0 < a\}\ z \leftarrow x;\ \{x = z \wedge y = x \wedge 0 < a\}$$

parbegin $x \leftarrow a * x \parallel y \leftarrow a * y$ **parend**

$$\{x = y\};$$

Most alkalmazhatjuk a diszjunkt párhuzamossági szabályt, és ezután a segédváltozót és a rá vonatkozó értékadást elhagyhatjuk.

DEFINÍCIÓ Legyen A az S programban szereplő egyszerű változók halmaza. Ha minden $x \in A$ csak $x \leftarrow e$ értékadásban fordul elő S-ben, ahol e ennek megfelelő kifejezés, akkor az A halmaz változóit segédváltozóknak nevezzük.

3.2. ÁLLÍTÁS A segédváltozók elhagyása nem változtatja meg az S program eredményét.

BIZONYÍTÁS Ha $x \in A$, akkor x nem szerepelhet olyan Boole-kifejezésben, amely befolyásolja az S programon átvezető utak meghatározását.

Ha $x \in A$, akkor x nem szerepelhet értékadás jobb oldalán sem, ha az értékadás nem segédváltozóra vonatkozik, ezért nem változtatja meg a program változóinak értékét, nem befolyásolja az állapotátmeneteket, amelyek nem segédváltozókra vonatkoznak.

SEGÉDVÁLTOZÓK AXIÓMÁJA:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S^{\#}\ \{q\}}$$

ahol az S program segédváltozókat tartalmazó program, az $S^{\#}$ pedig az a program, melyet az S programból a segédváltozók elhagyásával származtattunk.

Primitív műveletek és utasítások a meg nem szakítható műveletek ill. utasítások, amelyek csak egymás után hajtódhatnak végre.

Állapodjunk meg a következőkben:

Primitív (atomi) műveletek a következők:

- egy Boole-kifejezés kiértékelése;
- egy változó értékének a kiolvasása;
- értékadás egy változónak;
- egy olyan S programszakasz végrehajtása, amelyet primitív műveletnek tüntetünk fel a következő jelöléssel: ⟨S⟩, feltéve, hogy az S ciklust, vagy primitív utasítást nem tartalmazhat.

A primitív művelet következtetési szabálya mind parciális, mind teljes helyesség esetén a következő:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \langle S\rangle\ \{q\}}$$

3.4. PÉLDA Bizonyítsuk be a következő teljes helyességi tételt.

$\{n \geq 1\}$

$j \leftarrow 1; i \leftarrow 2; k_1 \leftarrow 0; k_2 \leftarrow 0;$

parbegin

while $j \leq n$ **do** $k_1 \leftarrow k_1 + j; j \leftarrow j + 2$ **od**;

||

while $i \leq n$ **do** $k_2 \leftarrow k_2 + i; i \leftarrow i + 2$ **od**;

parend;

$k \leftarrow k_1 + k_2;$

$\{k = (n * (n + 1)) / 2\}$

MEGOLDÁS Írjuk fel mindkét komponensre az iteráció invariánsát. Ehhez célszerű felhasználni a következő egyszerű összegfüggéseket:

Páros számok összege:

$$(2 + 4 + \ldots + 2 * l) = 2\,(1 + 2 + \ldots + l) = 2 * (l * (l + 1) / 2 = l * (l + 1);$$

Páratlan számok összege:

$$(1 + 3 + \ldots + (2 * l + 1)) = (1 + (2 + 1) + \ldots + (2 * l + 1) =$$

$$2\,(1 + 2 + \ldots + l) + (l + 1) = (l * (l + 1) + (l + 1) = (l + 1)^2;$$

Az invariánsok tehát a következők:

$$I_1(n; k_1, j): (1 \leq j \leq n) \wedge j = (2 * l + 1) \wedge k_1 = (l + 1)^2;$$

$$I_2(n; k_2, i): (2 \leq i \leq n) \wedge i = 2 * l \wedge k_2 = l * (l + 1);$$

Helyezzük el a megfelelő invariánsokat a program pontjaiban.

$\{n \geq 1\}$

$j \leftarrow 1; i \leftarrow 2; k_1 \leftarrow 0; k_2 \leftarrow 0;$

parbegin

$\{j = 1 \wedge k_1 = 0 \wedge n \geq 0 \wedge I_1(n; k_1, j)\}$

while $j \leq n$ **do** $k_1 \leftarrow k_1 + j; j \leftarrow j + 2$ **od**;

$\{(j =$ **if** páratlan(n) **then** n **else** $n + 1)$ **fi** $\wedge I_1(n; k_1, j)\}$

||

$\{i = 2 \wedge k_2 = 0 \wedge n \geq 0 \wedge I_2(n; k_2, i)\}$

while $i \leq n$ **do** $k_2 \leftarrow k_2 + i; i \leftarrow i + 2$ **od**;

$\{(i =$ **if** páros(n) **then** n **else** $n + 1)$ **fi** $\wedge I_2(n; k_2, i)\}$

parend;

$\{j =$ **if** páratlan(n) **then** n **else** $n + 1) \wedge I_1(n; k_1, j) \wedge$

$I_2(n; k_2, i) \wedge i =$ **if** páros(n) **then** n **else** $n + 1)$ **fi** $\}$

$k \leftarrow k_1 + k_2;$

$\{k = (n * (n + 1)) / 2\}$

Vizsgáljuk meg először, hogy a diszjunkt párhuzamossági szabály alkalmazható-e a feladat megoldására. Az első komponensre kapjuk, hogy

$$free(p_1) = free(j = 1 \wedge k_1 = 0 \wedge n \geq 0 \wedge I_1(n; k_1, j)) = \{k_1, j, n\};$$

$$change(S_1) = change(\textbf{while } j < n \textbf{ do } k_1 \leftarrow k_1 + j; j \leftarrow j + 2 \textbf{ od}) = \{k_1, j\};$$

$$free(q_1) = free(j = \textbf{if} \text{ páratlan}(n) \textbf{ then } n \textbf{ else } n + 1) \textbf{ fi} \wedge I_1(n; k_1, j)) = \{k_1, j, n\};$$

A második komponensre kapjuk, hogy

$$free(p_2) = free(i = 2 \wedge k_2 = 0 \wedge n \geq 0 \wedge I_2(n; k_2, i)) = \{k_2, i, n\};$$

$$change(S_2) = change(\textbf{while } i < n \textbf{ do } k_2 \leftarrow k_2 + i; i \leftarrow i + 2 \textbf{ od}) = \{k_2, i\};$$

$$free(q_2) = free(i = \textbf{if} \text{ páros}(n) \textbf{ then } n \textbf{ else } n + 1 \textbf{ fi} \wedge I_2(n; k_2, i)) = \{k_2, i, n\};$$

$$(\forall i, j \in \{1, 2\})(\text{free}(p_i) \cap \text{change}(S_j) = \{\} \wedge$$

$$\text{free}(q_i) \cap \text{change}(S_j) = \{k_1, j, n\} \cap \{k_2, i\} = \{\} \wedge$$

$$\{k_1, j, n\} \cap \{k_2, i\} = \{\} \wedge \{k_2, i, n\} \cap \{k_1, j\} = \{\} \wedge$$

$$\{k_2, i, n\} \cap \{k_1, j\} = \{\} = \{\});$$

Tehát alkalmazható a diszjunkt párhuzamossági szabály.

Bizonyítandó, hogy

$$(j = \textbf{if}\ \text{páratlan}(n)\ \textbf{then}\ n\ \textbf{else}\ n + 1\ \textbf{fi}) \wedge I_1(n; k_1, j) \wedge$$

$$(i = \textbf{if}\ \text{páros}(n)\ \textbf{then}\ n\ \textbf{else}\ n + 1\ \textbf{fi} \wedge I_2(n; k_2, i) \wedge k = k_1 + k_2) \Rightarrow$$

$$k = (n * (n + 1)) / 2,$$

azaz

$$(j = \textbf{if}\ \text{páratlan}(n)\ \textbf{then}\ n\ \textbf{else}\ n + 1\ \textbf{fi}) \wedge$$

$$(1 \leq j \leq n) \wedge j = (2 * l + 1) \wedge k_1 = (l + 1)^2 \wedge$$

$$(i = \textbf{if}\ \text{páros}(n)\ \textbf{then}\ n\ \textbf{else}\ n + 1\ \textbf{fi} \wedge$$

$$(1 \leq i \leq n) \wedge i = 2 * l \wedge k_2 = l * (l + 1) \wedge$$

$$k = k_1 + k_2) \Rightarrow k = (n * (n + 1)) / 2$$

Ezzel a tétel parciális helyességét bebizonyítottuk.

A ciklusok azonban befejeződnek, mivel

$$I_1(n; k_1, j) \Rightarrow j \leq n;$$

$$I_2(n; k_1, i) \Rightarrow i \leq n;$$

A diszjunkt párhuzamos programunk tehát teljesen helyes.

4. PÁRHUZAMOS PROGRAMOK KÖZÖS HASZNÁLATÚ VÁLTOZÓKKAL

Fogalmazzuk meg először a közös használatú változóhoz való hozzáférésre és a folyamatok sebességére vonatkozó követelményeket.

ALAPKÖVETELMÉNYEK

1. **A közös használatú változóhoz való hozzáférés követelménye**
   - Egynél több folyamat egyidejűleg nem férhet hozzá ugyanahhoz a közösen használt változóhoz.
   - A hozzáférésekre egymás után kerülhet csak sor.
   - A **hozzáférés** kétféle lehet:
     - a változó értékének a **kiolvasás**a;
     - **értékadás** a változónak.

     Mindkét hozzáférés **atomi utasítás**, ami azt jelenti, hogy más folyamat annak végrehajtását nem szakíthatja meg.
2. **A folyamatok sebességére vonatkozó követelmény**
   - Minden olyan folyamatnak a végrehajtási sebessége, amely
     - nem fejeződött be,
     - nem jutott holtpontra,

     tetszőleges pozitív érték, valahányszor a folyamat nem várakozik közösen használt változóhoz történő hozzáférésre.

- Ha a folyamat várakozásra kényszerül, akkor viszont garantált, hogy a folyamat a megkívánt hosszúságú időre hozzáférhet a közös használatú változóhoz.
- Ha egy művelet előfordulása végtelen sokszor megtörténhet egy végrehajtási sorozatban, akkor az végre is fog hajtódni.

4.1. PÉLDA

s ← 1; **parbegin** Q_1 || Q_2 **parend**;

Q_1: **while** "true" **do**

await s > 0 **then** s ← 0 **ta**; **await** "true" **then** s ← 1 **ta**; **od;**

Q_2: **while** "true" **do**

await s > 0 **then** s ← 0 **ta**; **await** "true" **then** s ← 0 **ta**; **od;**

Ha a fenti párhuzamos programban például először Q_1 kapja meg a vezérlést, akkor előállhat olyan helyzet, hogy az első ciklus magjának végrehajtása végtelen sokszor ismétlődhet, miközben Q_2 végrehajtására soha sem kerül sor. A fenti végtelen sorozat tehát feltevésünk szerint **nem legális** végrehajtási sorozat.

A LOGIKAI KIFEJEZÉS KIÉRTÉKELÉSÉNEK KÜLÖNLEGES SZEREPE A KONKURENS PROGRAMOZÁSBAN

- Szekvenciális programozás esetén a logikai kifejezés a programban az elágazás szerepét játssza, azaz kiértékelése alapján dől el, hogy egy utasítás végrehajtódik-e egy adott időpontban vagy sem. Például **if** α **then** S **fi**;
- Konkurens programozás esetén a logikai kifejezés általában szemaforszerepet játszik, azaz a folyamat továbbhaladását megállítja mindaddig,

amíg a logikai kifejezés értéke igaz nem lesz, amiről a folyamat ismétlődő kiértékeléssel vesz tudomást.

- Konkurens programozás esetén megvárakoztatásnál szükség lehet arra, hogy a közösen használt erőforrásra, a közösen használt változóra vonatkozó igényünket bejelentsük abból a célból, hogy előbb vagy utóbb sor kerüljön a beütemezésünkre. Ilyen igény bejelentésére szolgálhat az utasítás előtt elhelyezett logikai kifejezés, az őrfeltétel.

  Jelölése: $\langle \alpha \rightarrow S \rangle$;

  Például: $\langle s > 0 \rightarrow (s \leftarrow s - 1) \rangle$;

AZ ATOMI MŰVELETEK SZEREPE A KONKURENS PROGRAMOZÁSBAN

Konkurens programozás esetén szükség van annak megjelölésére, hogy egy utasítás, egy programszakasz úgy hajtódjon végre, hogy ne zavarja meg a vele párhuzamosan futó folyamatot, azaz egy összetett műveletet, utasítássorozatot atomi műveletté tegyünk. Ez a programszakasz azonban nem tartalmazhat iterációt. A meg nem szakítható utasításnak mindig be kell fejeződnie. **Az atomi utasítás jelölése**: $\langle S \rangle$;

Például: $\langle x \leftarrow x + 1 \rangle$; ill. $\langle x < 0 \rightarrow x \leftarrow x^2 \rangle$;

MEGJEGYZÉS További jelölések:

Meg nem osztható (atomi) utasítások más szokásos jelölései:

$\langle S \rangle$;: **await** "true" **then** S **ta**;

$\langle \alpha \rightarrow S \rangle$;: **await** α **then** S **ta**;

$\langle \alpha \rightarrow$ **skip**$\rangle$;: **wait** α;

Párhuzamos programszerkezet más szokásos jelölései:

$[S_1 \parallel \ldots \parallel S_n]$;: **parbegin** $S_1 \parallel \ldots \parallel S_n$ **parend**;

$[S_1 \parallel \ldots \parallel S_n]$;: **cobegin** $S_1 \parallel \ldots \parallel S_n$ **coend**;

FOLYAMATOK ABSZTRAKT NYELVI ÁBRÁZOLÁSAI KÖZÖS HASZNÁLATÚ VÁLTOZÓK ESETÉN

Az általános programozási nyelv absztrakciójának szintaxisa legyen a következő:

utasítás: $S ::= \alpha \rightarrow y \leftarrow f(x, y) \mid S_1; S_2 \mid$ **if** $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **fi** $\mid$
do $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **od** $\mid$
do $S_B \square S_e$; **exit od** $\mid$
parbegin $S_1 \parallel S_2$ **parend**;

if $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **fi** nem determinisztikus kiválasztás;

do $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **od** nem determinisztikus iteráció;

do $S_B \square S_e$; **exit od** kilépéses nem determinisztikus iteráció, ahol S_B az iteráció $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ magjának a jelölése.

A nyelv kiterjesztése párhuzamos programok atomi utasításaival:

utasítás: $S ::= \alpha \rightarrow y \leftarrow f(x, y) \mid S_1; S_2 \mid$ **if** $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **fi** $\mid$
do $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **od** $\mid$
do $S_B \square S_e$; **exit od** $\mid$
parbegin $S_1 \parallel S_2$ **parend**;
meg nem szakítható utasítás: $\langle S \rangle$;

MEGÁLLAPODÁS A **parbegin** $S_1 \parallel S_2$ **parend** párhuzamos struktúra végrehajtása során meg nem szakítható (atomi) műveletek a következők:

- α logikai kifejezés kiértékelése;
- egy közös változó értékéhez való hozzáférés;

- a **skip** végrehajtása;
- az ⟨S⟩ programszakasz végrehajtása.

EGYSZERŰ KRITIKUS HIVATKOZÁSOK

- közös változó értékének a kiolvasása.
- közös változónak történő értékadás.

A közös változó lehet egyszerű vagy indexes változó.

(A hardver csak az egyszerű kritikus hivatkozásoknak a meg nem szakíthatóságát (atomi voltát) garantálja.)

KRITIKUS SZAKASZNAK nevezünk egy olyan programszakaszt, amelyben olyan változó értéke változik meg, amelynek szerepe van a párhuzamos program komponensei közötti együttműködésnek, a szinkronizációnak a definiálásában.

KÖLCSÖNÖS KIZÁRÁS Kölcsönös kizárás esetén azt követeljük meg, hogy a párhuzamos komponensek algoritmusai ne változtassák meg azoknak a változóknak az értékeit, amelyek a kritikus szakaszban a folyamatok közötti szinkronizáció megvalósítására szolgálnak.

4.2. PÉLDA Meg nem szakítható utasításokat tartalmazó program:

if $\langle x^2 < y \rightarrow x \leftarrow x + 1 \rangle; \langle y \leftarrow y - 1 \rangle \square \langle x^2 \geq y \rightarrow$ **skip**$\rangle$ **fi;**

A fenti program megszakítható utasításokat tartalmazó formában:

if $x^2 < y \rightarrow x \leftarrow x + 1; y \leftarrow y - 1 \square x^2 \geq y \rightarrow$ **skip fi**;

Az $x^2 < y \rightarrow x \leftarrow x + 1$ forma az $\langle x^2 < y \rightarrow$ **skip**$\rangle$; $x \leftarrow x + 1$; program rövidített változata.

4.3. PÉLDA A Dijkstra-féle szemaforutasítások implementációja:

P(s): $\langle s > 0 \rightarrow s \leftarrow s - 1 \rangle$;

V(s): $\langle$"true" $\rightarrow s \leftarrow s + 1 \rangle$;

Ezek azonban nem pontos implementációk. A pontos implementációra, majd később visszatérünk.

A PRIMITÍV PROGRAM KÖVETKEZTETÉSI SZABÁLYA mind parciális, mind teljes helyesség esetén a következő:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \langle S\rangle\ \{q\}}$$

DEFINÍCIÓ Primitív programokra vonatkozó tételek **interferenciamentességének definíciója.** Legyen adva a párhuzamos programra vonatkozó teljes helyességi tétel:

$$\{p\}\ \textbf{parbegin}\ \langle S_1\rangle \parallel \ldots \parallel \langle S_n\rangle\ \textbf{parend}\ \{q\}$$

és annak bizonyítása.

A bizonyításban szereplő $\{p_i\}\ \langle S_i^*\rangle\ \{q_i\}$ és $\{p_j\}\ \langle S_j^*\rangle\ \{q_j\}$, $(i, j \in \{1, \ldots, n\})$ tételeket interferenciamentes tételeknek nevezzük, ha a következők teljesülnek:

$$(\forall i \neq j,\ i, j \in \{1, 2\})(\{p_i \wedge p_j\}\ \langle S_j^*\rangle\ \{p_i\} \text{ és } \{q_i \wedge p_j\}\ \langle S_j^*\rangle\ \{q_i\}).$$

Itt S^* ún.. annotációs program, ami az S programnak a tétel bizonyításához szükséges állításokkal bővített változatát jelenti.

DEFINÍCIÓ Legyen megadva a párhuzamos program S komponense, a $\{p\}\ S\ \{q\}$ parciális helyességi tétel és annak $\{p\}\ S^*\ \{q\}$ bizonyítása, továbbá az u értékadó utasítás a pre_u előfeltétellel. Azt mondjuk, hogy **az u értékadó utasítás az adott előfeltétel mellett nem interferál a** $\{p\}\ S^*\ \{q\}$ **parciális helyességi tétellel**, ha minden $\{p'\}\ S'\ \{q'\}$ tételre, amely a $\{p\}\ S^*\ \{q\}$ bizonyításában szerepel, igaz, hogy

$\{pre_u \wedge p'\}\ u\ \{p'\}$ **és** $\{pre_u \wedge q'\}\ u\ \{q'\}$.

DEFINÍCIÓ Legyen megadva a

$$\{p\}\ \textbf{parbegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{parend}\ \{q\}$$

tétel, és az annak bizonyításához felhasznált $\{p_i\}\ S_i^*\ \{q_i\}$, $i \in \{1, \ldots, n\}$ tételek. Tekintsük S_i-ben az összes értékadó utasítást és primitív utasítássá tett programszakaszt. Jelöljön egy ilyen utasítást u_i és annak előfeltételét pre_{u_i}

Ha minden $i \in \{1, \ldots, n\}$ estén

$$\{pre_{u_j}\}\ u_i,\ \textbf{nem interferál a}\ \{p_j\}\ S_j^*\ \{q_j\}$$

tétellel valahányszor i, $j \in \{1, \ldots, n\}$ és $j \neq i$, akkor azt mondjuk, hogy a $\{p_i\}\ S_i^*\ \{q_i\}$, $i \in \{1, \ldots, n\}$, **tételek interferenciamentesek**.

KÖZÖS VÁLTOZÓKKAL RENDELKEZŐ PÁRHUZAMOS PROGRAMOK KÖVETKEZTETÉSI SZABÁLYA:

$$\frac{\{p_i\}\ S_i^*\ \{q_i\},\ i \in \{1, \ldots, n\}\ \text{tételek igazak és interferenciamentesek}}{\{p_1 \wedge \ldots \wedge p_n\}\ \textbf{parbegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{parend}\ \{q_1 \wedge \ldots \wedge q_n\}}$$

Általánosabb formában:

$$\frac{\begin{array}{c}\{p_i\}\ S_i^*\ \{q_i\},\ i \in \{1, \ldots, n\}\ \text{tételek igazak és interferenciamentesek és} \\ (p \Rightarrow p_1 \wedge \ldots \wedge p_n)\ \textbf{és} \\ q_1 \wedge \ldots \wedge q_n \Rightarrow q\end{array}}{\{p\}\ \textbf{parbegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{parend}\ \{q\}}$$

TELJES HELYESSÉG BIZONYÍTÁSA Teljes helyesség bizonyításánál mind az állítások, mind a befejezés garantálásához használt függvény esetében vizsgálni kell az interferenciamentességet.

DEFINÍCIÓ Legyen adva a $\{p\}\ S\ \{q\}$ teljes helyességi tétel és annak $\{p\}\ S^*\ \{q\}$ bizonyítása, továbbá az u értékadó utasítás a pre_u előfeltétellel. Azt mondjuk, hogy **az u értékadó utasítás az adott előfeltétel mellett nem interferál a** $\{p\}\ S^*\ \{q\}\}$ **teljes helyességi tétellel**, ha minden $\{p'\}\ S'\ \{q'\}$ tételre, amely a $\{p\}\ S^*\ \{q\}$ bizonyításában teljes helyességi értelemben szerepel, igaz, hogy

- $\{pre_u \wedge p'\}\ u\ \{p'\}$ **és** $\{pre_u \wedge q'\}\ u\ \{q'\}$;
- $\{pre_u \wedge E_0 = E\}\ u\ \{E' \leq E_0\}$.

DEFINÍCIÓ Adott a $\{p\}$ **parbegin** $S_1 \parallel \ldots \parallel S_n$ **parend** $\{q\}$ teljes helyességi tétel, és az annak bizonyításához felhasznált $\{p_i\}\ S_i^*\ \{q_i\}$, $i \in \{1, \ldots, n\}$, teljes helyességi tételek. Tekintsük S_i-ben az összes értékadó utasítást és primitív utasítássá tett programszakaszt. Jelöljön egy ilyen utasítást u_j és annak előfeltétele legyen pre_{ui}. Ha minden $i \in \{1, \ldots, n\}$ estén $\{pre_{ui}\}\ u_i$ **nem interferál** a $\{p_j\}\ S_j^*\ \{q_j\}$ tétellel teljes helyességi értelemben, valahányszor $j \in \{1, \ldots, n\}$ és $j \neq i$, akkor azt mondjuk, hogy a $\{p_i\}\ S_i^*\ \{q_i\}$, $i \in \{1, \ldots, n\}$ **tételek interferenciamentesek**.

KÖVETKEZTETÉSI SZABÁLYOK

Értékadás következtetési szabálya:

$$\frac{(p(x, y) \wedge \alpha(x, y)) \Rightarrow q(x, g(x, y))}{\{p(x, y)\}\ \alpha(x, y) \rightarrow y \leftarrow g(x, y)\ \{q(x, y)\}}$$

Következmény szabálya:

$$\frac{p \Rightarrow p_0,\ \{p_0\}\ S\ \{q_0\},\ q_0 \Rightarrow q}{\{p\}\ S\ \{q\}}$$

Szekvencializáció szabálya:

$$\frac{p \Rightarrow p_1;\ \{p_1\}\ S_1^*\ \{r_1\};\ r_1 \Rightarrow r_2;\ \{r_2\}\ S_2^*\ \{q_2\};\ q_2 \Rightarrow q}{\{p\}\ S_1; S_2\ \{q\}}$$

Nem determinisztikus kiválasztás szabálya:

$$\frac{\{p\}\ S_j^*\ \{q\},\ \forall j \in \{1, \ldots, n\}}{\{p\}\ \textbf{if}\ \alpha_1 \rightarrow S_1 \,\square\, \ldots \,\square\, \alpha_n \rightarrow S_n\ \textbf{if}\ \{q\}}$$

Kilépéses nem determinisztikus iteráció szabálya:

$$\frac{\{I\}\ S_B^*\ \{I\},\ \{I\}\ S_e^*\ \{q\}}{\{I\}\ \textbf{do}\ S_B \,\square\, S_e;\ \textbf{exit od}\ \{q\}}$$

Párhuzamos kompozíció következtetési szabálya:

$$\frac{\{p_i\}\ S_i^*\ \{q_i\},\ i \in \{1, \ldots, n\}\ \text{tételek igazak } \textbf{és}\ \text{interferenciamentesek}}{\{p_1 \wedge \ldots \wedge p_n\}\ \textbf{parbegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{parend}\ \{q_1 \wedge \ldots \wedge q_n\}}$$

A determinisztikus iteráció következtetési szabályának szokásos formája:

$$P(x, y) \Rightarrow I(x, y, 0);$$

$$\{\{I(x, y, j) \wedge \alpha(x, y) \wedge j = j_0\}\}\ S\ \{\{I(x, y) \wedge j < j_0\}\};$$

$$I(x, y, j) \Rightarrow 0 \leq j;$$

$$I(x, y, j) \wedge \neg\alpha(x, y) \Rightarrow q(x, y);$$

$$\{\{p(x, y)\}\}\ \textbf{while}\ \alpha(x, y)\ \textbf{do}\ S\ \textbf{od}\ \{\{q(x, y)\}\}$$

$j \in \{0, 1, 2, \ldots\}$.

4.4. PÉLDA Bizonyítsuk be a következő tételt.

$$\{x = 0\}\ \textbf{parbegin}\ \langle x \leftarrow x + 2\rangle \parallel \langle x \leftarrow x + 3\rangle\ \textbf{parend}\ \{x = 5\}$$

BIZONYÍTÁS Az értékadás következtetési szabálya alapján nyilvánvalóan igazak a következő tételek:

$$\{x = 0 \vee x = 3\}\ x \leftarrow x + 2\ \{x = 2 \vee x = 5\},$$

$$\{x = 0 \vee x = 2\}\ x \leftarrow x + 3\ \{x = 3 \vee x = 5\}.$$

A tételek interferenciamentességéhez a következőket kell bizonyítani:

$$\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 3)\}\ x \leftarrow x + 3\ \{\ x = 0 \vee x = 3\},$$

$$\{(x = 0 \vee x = 2) \wedge (x = 2 \vee x = 5)\}\ x \leftarrow x + 3\ \{\ x = 2 \vee x = 5\},$$

illetve

$$\{(x = 0 \vee x = 3) \wedge (\ x = 0 \vee x = 2)\}\ x \leftarrow x + 2\{\ x = 0 \vee x = 2\},$$

$$\{(x = 0 \vee x = 3) \wedge (\ x = 3 \vee x = 5)\}\ x \leftarrow x + 2\{\ x = 3 \vee x = 5\},$$

Ezek után – a következtetési szabály alkalmazásához – még bizonyítandók az indulásra és a befejezésre vonatkozó feltételek:

$$(x = 0 \Rightarrow (x = 0 \vee x = 3)) \wedge (x = 0 \Rightarrow (x = 0 \vee x = 2))$$

$$((x = 2 \vee x = 5) \wedge (x = 3 \vee x = 5\ )) \Rightarrow x = 5.$$

4.5. PÉLDA Bizonyítsuk be a következő tételt:

$$\{x = 0\}\ \textbf{parbegin}\ x \leftarrow x + 2 \parallel x \leftarrow 0\ \textbf{parend}\ \{x = 0 \vee x = 2\}$$

A lehetséges végrehajtások eredménye:

$$\{x = 0\}\ x \leftarrow x + 2;\ x \leftarrow 0\ \{x = 0\},$$

$$\{x = 0\}\ x \leftarrow 0;\ x \leftarrow x + 2\ \{x = 2\}.$$

De végezzük el a formális bizonyítást!

BIZONYÍTÁS

$$\{x = 0\}\ x \leftarrow x + 2\ \{x = 2\},$$

$$\{\text{"true"}\}\ x \leftarrow 0\ \{x = 0\},$$

a baj az, hogy

$$x = 2 \wedge x = 0 \equiv \text{"false"} \Rightarrow x = 2 \vee x = 0.$$

A problémát az okozza, hogy a fenti tételek helyes tételek, de az interferenciamentesség nem teljesül:

$$\{\text{"true"}\}\ x \leftarrow 0\ \{x = 0\},$$

$$\{x = 0\}\ x \leftarrow x + 2\ \{x = 2\},$$

$\{x = 2 \wedge \text{"true"}\}\ x \leftarrow 0\ \{x = 0\}$; tételnél $x = 2$ nem őrződik meg.

Ha a bizonyításnál helyesen egészítjük ki a részeredményeket, akkor már teljesül az interferenciamentesség

$$\{\text{"true"}\}\ x \leftarrow 0\ \{x = 0 \vee x = 2\},$$

$$\{x = 0\}\ x \leftarrow x + 2\ \{x = 2 \vee x = 0\},$$

és a tétel helyessége is bebizonyítható.

A másik járható út az, ha segédváltozókat alkalmazunk. Vezessünk be két segédváltozót: $kész_1$ értéke legyen igaz, ha az első komponens végrehajtódott, legyen hamis különben. Hasonlóan definiáljuk $kész_2$ értékét is. Készítsük el a programunkat ezekkel a segédváltozókkal, beleírva a helyesség bizonyításához szükséges invariánsokat is.

$\{x = 0\}$ **parbegin** $x \leftarrow x + 1 \parallel x \leftarrow 0$ **parend** $\{x = 1 \vee x = 0\}$

Az annotált program: $\{x = 0\}$

$kész_1 \leftarrow$ "false"; $kész_2 \leftarrow$ "false"; $x \leftarrow 0$;

$\{kész_1 =$ "false" $\wedge\ kész_2 =$ "false" $\wedge\ x = 0\}$

parbegin

$\{kész_1 =$ "false" $\wedge\ x = 0\}$

$(x, kész_1) \leftarrow (x + 1,$ "true");

$\{(kész_1 =$ "true" $\wedge\ (x = 0 \vee x = 1))\}$

$\parallel$

$\{kész_2 =$ "false" $\wedge\ (kész_1 =$ "false" $\Rightarrow x = 0 \wedge kész_1 =$ "true" $\Rightarrow x = 1)\}$

$(x, kész_2) \leftarrow (0,$ "true")

$\{(kész_2 =$ "true" $\wedge$

$(kész_1 =$ "false" $\Rightarrow x = 0 \wedge kész_1 =$ "true" $\Rightarrow (x = 0 \vee x = 1))\}$

parend;

$\{(kész_1 =$ "true" $\wedge\ kész_2 =$ "true" $\Rightarrow (x = 0 \vee x = 1))\}$

$(kész_1 =$ "true" $\wedge\ (x = 0 \vee x = 1)) \wedge (kész_2 =$ "true" $\wedge$

$(kész_1 =$ "false" $\Rightarrow x = 0 \wedge kész_1 =$ "true") $\Rightarrow (x = 0 \vee x = 1))$

Ha az invariánsokat helyesen írtuk fel, akkor a helyesség bizonyítása egyszerű feladat. Nézzük ezt a példát általánosan.

4.6. PÉLDA. Adjuk meg a következő program eredményét és bizonyítsuk be annak helyességét.

$\{x = a \wedge y = b\}$**parbegin** $\langle x \leftarrow f(x, y)\rangle \parallel \langle y \leftarrow g(x, y)\rangle$ **parend** $\{?\}$

BIZONYÍTÁS A lehetséges végrehajtások:

$\{x = a \wedge y = b\}\ y \leftarrow g(x, y);\ x \leftarrow f(x, y)\ \{x = f(a, g(a, b)) \wedge y = g(a, b)\};$

$\{x = a \wedge y = b\}\ x \leftarrow f(x, y);\ y \leftarrow g(x, y)\ \{x = f(a, b) \wedge y = g(f(a, b), b)\}.$

Az eredmény tehát a következő

$$\{x = a \wedge y = b\}$$

$$\textbf{parbegin}\ \langle x \leftarrow f(x, y)\rangle \parallel \langle y \leftarrow g(x, y)\rangle\ \textbf{parend}$$

$$\{[x = f(a, g(a, b)) \wedge y = g(a, b)] \vee [x = f(a, b) \wedge y = g(f(a, b), b)]\}.$$

Végezzük el ennek a tételnek a bizonyítását segédváltozók bevezetésével. Ismét vezessük be az előbbi példához hasonló céllal a $kész_1$, $kész_2$ segédváltozókat. Ezekkel a segédváltozókkal az annotált program a következő lesz:

$$\{x = a \wedge kész_1 = \text{"false"} \wedge y = b \wedge kész_2 = \text{"false"}\}$$

$$\textbf{parbegin}$$

$$\{kész_1 = \text{"false"} \wedge (kész_2 = \text{"false"} \Rightarrow (x = a \wedge y = b) \wedge$$

$$kész_2 = \text{"true"} \Rightarrow (x = a \wedge y = g(a, b))\}$$

$$(x, kész_1) \leftarrow (f(x, y), \text{"true"})$$

$$\{(kész_1 = \text{"true"} \wedge (kész_2 = \text{"false"} \Rightarrow (x = f(a, b) \wedge y = b) \wedge$$

$$kész_2 = \text{"true"} \Rightarrow (x = f(a, g(a, b)) \wedge y = g(a, b)) \vee$$

$$(x = f(a, b) \wedge y = g(f(a, b), b)))\}$$

$$\parallel$$

$$\{kész_2 = \text{"false"} \wedge (kész_1 = \text{"false"} \Rightarrow (x = a \wedge y = b) \wedge$$

$$kész_1 = \text{"true"} \Rightarrow (x = f(a, b) \wedge y = b))\}$$

$$(y, kész_2) \leftarrow (g(x, y), \text{"true"})$$

$$\{(kész_2 = \text{"true"} \wedge (kész_1 = \text{"false"} \Rightarrow (x = a \wedge y = g(a, b)) \wedge$$

$$kész_1 = \text{"true"} \Rightarrow (x = f(a, g(a, b)) \wedge y = g(a, b)) \vee$$

$$(x = f(a, b) \wedge y = g(f(a, b), b)))\}$$

$$\textbf{parend}$$

$$\{[x = f(a, g(a, b)) \wedge y = g(a, b)] \vee [x = f(a, b) \wedge y = g(f(a, b), b)]\}.$$

A segédváltozók azért hasznosak, mert segítenek abban, hogy a részeredményeket ne hibázzuk el. A segédváltozókat azonban el is hagyhatjuk a bizonyításnál, ha a részeredményeket helyesen írjuk fel. Ebben az általános esetben ez a következőképpen történhet:

$$\{x = a \wedge y = b\}$$

$$\textbf{parbegin } \langle x \leftarrow f(x, y)\rangle \parallel \langle y \leftarrow g(x, y)\rangle \textbf{ parend}$$

$$\{[x = f(a, g(a, b)) \wedge y = g(a, b)] \vee [x = f(a, b) \wedge y = g(f(a, b), b)]\}.$$

Jelölje az eredményt q:

$$[x = f(a, g(a, b)) \wedge y = g(a, b)] \vee [x = f(a, b) \wedge y = g(f(a, b), b)].$$

Bizonyítsuk be a tétel helyességét.

Vagy $x \leftarrow f(x, y)$ az első a végrehajtás során, vagy előtte lefutott már a másik. Így

$$\{(x = a \wedge y = b) \vee (x = a \wedge y = g(a, b))\}$$

$$x \leftarrow f(x, y)$$

$$\{[x = f(a, b) \wedge y = b] \vee [x = f(a, g(a, b)) \wedge y = g(a, b)]\}.$$

Hasonlóképpen

$$\{[(x = a \wedge y = b)] \vee [\, x = f(a, b) \wedge y = b]\}$$

$$y \leftarrow g(x, y)$$

$$\{[x = a \wedge y = g(a, b)] \vee [x = f(a, b) \wedge y = g(f(a, b), b)]\}.$$

Sajnos ezek a tételek, bár helyesek, nem interferenciamentesek, mert p_2 ugyan változatlan marad, de q_2 nem őrződik meg.

$$\{[(x = a \wedge y = b) \vee (x = a \wedge y = g(a, b))]_{p_1} \wedge [(x = a \wedge y = b) \vee (x = f(a, b) \wedge y = b)]_{p_2}\}$$

$$x \leftarrow f(x, y)$$

$$\{(x = a \wedge y = b) \vee (x = f(a, b) \wedge y = b)\}_{p_2},$$

$$\{[(x = a \wedge y = b) \vee (x = a \wedge y = g(a, b))]_{p_1} \wedge [(x = a \wedge y = g(a, b)) \vee (x = f(a, b) \wedge y = g(f(a, b), b))]_{q_2}\}$$

$$x \leftarrow f(x, y)$$

$$\{x = f(a, g(a, b)) \wedge y = g(a, b)\}.$$

Hasonló a helyzet q_1 esetében is. Ezért legyen

$$q_1^*: (x = f(a, b) \wedge y = b) \vee (x = f(a, g(a, b)) \wedge y = g(a, b)) \vee (x = f(a, b) \wedge y = g(f(a, b), b)),$$

$$q_2^*: (x = a \wedge y = g(a, b)) \vee (x = f(a, g(a, b)) \wedge y = g(a, b)) \vee (x = f(a, b) \wedge y = g(f(a, b), b)).$$

Ekkor már teljesül az interferenciamentesség, ugyanis

$$\{(x = a \wedge y = b) \vee (x = a \wedge y = g(a, b))\}$$

$$x \leftarrow f(x, y)$$

$$\{(x = f(a, b) \wedge y = b) \vee (x = f(a, g(a, b)) \wedge y = g(a, b)) \vee (x = f(a, b) \wedge y = g(f(a, b), b))\}$$

Hasonlóképpen

$$\{[(x = a \wedge y = b)] \vee [x = f(a, b) \wedge y = b]\}$$

$$y \leftarrow g(x, y)$$

$$\{(x = a \wedge y = g(a, b)) \vee (x = f(a, g(a, b)) \wedge y = g(a, b)) \vee (x = f(a, b) \wedge y = g(f(a, b), b))\}.$$

Ezek a tételek tehát igazak és interferenciamentesek, másrészt

$$q_1^* \wedge q_2^* \Rightarrow q = (x = f(a, g(a, b)) \wedge y = g(a, b)) \vee (x = f(a, b) \wedge y = g(f(a, b), b)).$$

5. PÁRHUZAMOS PROGRAMOK SZINKRONIZÁCIÓVAL

Szükségünk van egy absztrakt nyelvre, amelyben olyan párhuzamos programokat írhatunk le, amelyek szinkronizációs feladatok megoldására is alkalmasak.

Terjesszük tehát ki eddigi programozási nyelvünket ilyen problémák megoldására is. Ehhez a korábban használt párhuzamos szerkezetek mellett őrfeltételekkel ellátott atomi utasításokra is szükségünk van. Legyen a programozási nyelvünk nem determinisztikus programozási nyelv, amelynek szintaxisa a következő:

utasítás: $S ::= \alpha \rightarrow y \leftarrow f(x, y) \mid S_1; S_2 \mid \alpha \rightarrow S \mid$

if $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **fi** |

do $\alpha_1 \rightarrow S_1 \square \ldots \square \alpha_n \rightarrow S_n$ **od** |

do $S_B \square S_e$; **exit od** |

parbegin $S_1 \parallel S_2$ **parend**;

meg nem szakítható utasítás: $\langle S \rangle$;

A meg nem szakítható

$\langle \alpha \rightarrow S \rangle$; **: await** α **then** S **ta**;

és az

$\langle S \rangle$; **: await** "true" **then** S **ta**;

utasításokban

- α kvantorfüggetlen logikai kifejezés;
- S determinisztikus program, amely iterációt és megvárakoztató utasítást nem tartalmazhat.

Az **await** α **then** S **ta** megvárakoztató utasítás jelentése:

1. Mindaddig, amíg α kiértékelése "false" értéket eredményez, az utasítás végrehajtása felfüggesztődik.
2. Ha α kiértékelése "true" értéket eredményez, az S program a kiértékeléssel együtt meg nem szakítható utasításként hajtódik végre.

A megvárakoztató utasítás speciális esetei:

- **await** "true" **then** S **ta** $\equiv$ <S>;

  Például: {"true"}

  await "true" **then if** $x \neq y$ **then** $x \leftarrow y + 1$ **fi ta**

  $\{x = y \vee x = y + 1\}$

- **await** α **then** skip **ta** $\equiv$ <$\alpha \rightarrow$ skip>;

A Dijkstra-féle szemaforutasítások implementálása:

$$Ds = (\{S, integer\}, \{init, P, V\}).$$

$$init: integer \rightarrow S;$$

$$P: S \times integer \rightarrow S;$$

$$V: S \times integer \rightarrow S;$$

Az utasítások jelentése.

P(s): $s \leftarrow s - 1$; **if** $s < 0$ **then** állj be az s-hez rendelt várakozó sorba **fi**;

V(s): $s \leftarrow s + 1$; **if** $s \leq 0$ **then** induljon el egy folyamat az s- hez rendelt sorból **fi**;

Egyszerű implementációjuk megvárakoztató utasítással:

P(s): **await** s > 0 **then** s ← s - 1 **ta**;

V(s): **await** "true" **then** s ← s + 1 **ta**;

Ez a megvalósítás szemantikailag nem teljesen azonos a fenti definícióval. Ott előbb történik az s értékének csökkentése és ezt követi a kiértékelés. Ha az s értéke negatív, akkor a szemafor abszolút értéke a szemafornál várakozó folyamatok számával azonos. Itt ez sem teljesül.

A szemaforutasítások egy pontos megvalósítása lehet például a következő, ha eltekintünk attól a szabályunktól, hogy az atomi utasításon belül nem helyezhetünk el megvárakoztató utasítást, mert az atomi utasításnak be kell fejeződnie.

```
P(s): await "true" then begin
        s ← s - 1; s < 0 → V[j] ← "false";
        await V[j] then skip ta end ta
```

```
V(s): await "true" then begin
        s ← s + 1; if s ≤ 0 then
        begin válassz egy olyan i-t, amelyre V[i] = "false";
        V[i] ← "true" end
                fi end ta;
```

DEFINÍCIÓ A párhuzamos program valamelyik ága **blokkolt állapot**ba kerül, ha az abban lévő valamelyik

$$\{p\}\ \textbf{await}\ \alpha\ \textbf{then}\ S\ \textbf{ta}$$

szinkronizációs utasítás végrehajtása előtt $p \wedge \alpha$ = "false".

DEFINÍCIÓ Azt mondjuk, hogy a párhuzamos program **holtpontállapot**ba jutott, ha a párhuzamos program nem fejeződött be, és nincs olyan utasítása, amely megkapta a vezérlést és végre is tudja azt hajtani.

A megvárakoztató utasítás szavakban megfogalmazott szemantikáját formálisan a következő tranzakciós szabály definiálhatja:

$$\frac{\langle S, \sigma\rangle \rightarrow \langle E, \sigma'\rangle}{\langle \textbf{await } \alpha \textbf{ then } S \textbf{ ta} \rangle \rightarrow \langle E, \sigma'\rangle}$$

Ha α = "false", nincs tranzakció.

DEFINÍCIÓ Legyen S párhuzamos program, amelynek σ állapotát a p állítás határozza meg.

- Az $\langle S, \sigma\rangle$ konfigurációról azt mondjuk, hogy **holtpont-konfiguráció**, ha $\sim(S \equiv E)$ és az $\langle S, \sigma\rangle$ konfigurációnak nincs rákövetkező konfigurációja a $\rightarrow$ tranzakciós reláció szerint.
- Az S programról azt mondjuk, hogy σ állapotból **holtpontra juthat**, ha létezik olyan tranzakciós sorozat a σ állapotból kiindulva, amelyik holtpont-konfigurációra vezet.
- Az S programról azt mondjuk, hogy p állítás szerint **holtpontmentes**, ha nem létezik olyan, a p állításnak eleget tevő σ állapot, amelyből a tranzakciós sorozat holtpont-konfigurációra vezethet.

AZ S PÁRHUZAMOS PROGRAM SZEMANTIKÁJA: $M[S]: \Sigma \rightarrow \Sigma'$.

DEFINÍCIÓ Legyenek $\perp$, Δ és fail **virtuális állapotok**, amelyek jelentése a következő:

$\perp$: üres eredmény, a be nem fejeződő program eredménye.

Δ: a holtpont miatt be nem fejeződő program üres eredménye.

fail: a végrehajtás során előálló hibás eredmény, pl. túlcsordulás.

PARCIÁLIS HELYESSÉGI SZEMANTIKA

$M[S](\sigma) = \{\sigma' \mid <S, \sigma> \rightarrow <E, \sigma'>\}$. (Feltevés $\sigma' \notin \{\perp, \Delta, \text{fail}\}$.

GYENGE TELJES HELYESSÉGI SZEMANTIKA

$M_{wtot}[S](\sigma) = M[S](\sigma) \cup \{\perp\}$. (Feltevés $\sigma' \notin \{\Delta, \text{fail}\}$.

TELJES HELYESSÉGI SZEMANTIKA

$M_{tot}[S](\sigma) = M_{wtot}[S](\sigma) \cup \{\Delta\}$.(Feltevés $\sigma' \notin \{\text{fail}\}$.

A SZINKRONIZÁCIÓS UTASÍTÁS KÖVETKEZTETÉSI SZABÁLYA:

$$\frac{\{p \wedge \alpha\}\ S\ \{q\}}{\{p\}\ \textbf{await}\ \alpha\ \textbf{then}\ \{p \wedge \alpha\}\ S^*\ \{q\}\ \textbf{ta}\ \{q\}}$$

Mind parciális, mind teljes helyesség esetén ez a szabály érvényes.

5.1. PÉLDA $\{x = a\}$ **await** $x = b$ **then** skip **ta** $\parallel x \leftarrow b\ \{x = b\}$

BIZONYÍTÁS $\{x = a \vee x = b\}$ **await** $x = b$ **then** skip **ta** $\{x = b\}$

$$\{\text{"true"}\}\ x \leftarrow b\ \{x = b\}$$

$$\{(x = a \vee x = b) \wedge \text{"true"}\}\ x \leftarrow b\ \{x = a \vee x = b\}$$

$$\{x = b \wedge \text{"true"}\}\ x \leftarrow b\ \{x = b\}$$

$$\{(x = a \vee x = b) \wedge x = b\}\ \textbf{await}\ x = b\ \textbf{then}\ \text{skip}\ \textbf{ta}\ \{x = b\}$$

$$\{(x = a \vee x = b) \wedge \text{"true"}\}\ \textbf{await}\ x = b\ \textbf{then}\ \text{skip}\ \textbf{ta}\ \{\text{"true"}\}$$

Gyenge teljes helyesség: A definíció alapján a $\{p\}\ S\ \{q\}$ tétel gyenge teljes helyességi értelemben teljesül, ha

$$p(\sigma) \Rightarrow M_{wtot}[S](\sigma) \subseteq \{\tau \mid q(\tau)\}.$$

Mivel $\sim(\perp \in \{\tau \mid q(\tau)\})$, az S végrehajtása véges, ezért vagy befejeződik úgy, hogy q teljesül, vagy blokkolt állapotba kerül.

DEFINÍCIÓ Legyen adott a

$$\{p\}\ \textbf{parbegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{parend}\ \{q\}$$

gyenge teljes helyességi tétel és az annak bizonyításához felhasznált $\{p_i\}\ S_i^*\ \{q_i\}$, $i \in \{1, \ldots, n\}$ gyenge teljes helyességi tételek. Tekintsük S_i-ben az összes értékadó utasítást, és az összes **await** utasítást. Jelöljön egy ilyen utasítást u_i, annak előfeltételét pedig pre_{ui}.

Ha minden $i \in \{1, \ldots, n\}$ estén

$$\{pre_{ui}\}\ u_i,\ \textbf{nem interferál a}\ \{p_j\}\ S_j^*\ \{q_j\}$$

tétellel valahányszor $j \in \{1, \ldots, n\}$ és $j \neq i$, akkor azt mondjuk, hogy a $\{p_i\}\ S_i^*\ \{q_i\}$, $i \in \{1, \ldots, n\}$, **tételek interferenciamentesek**.

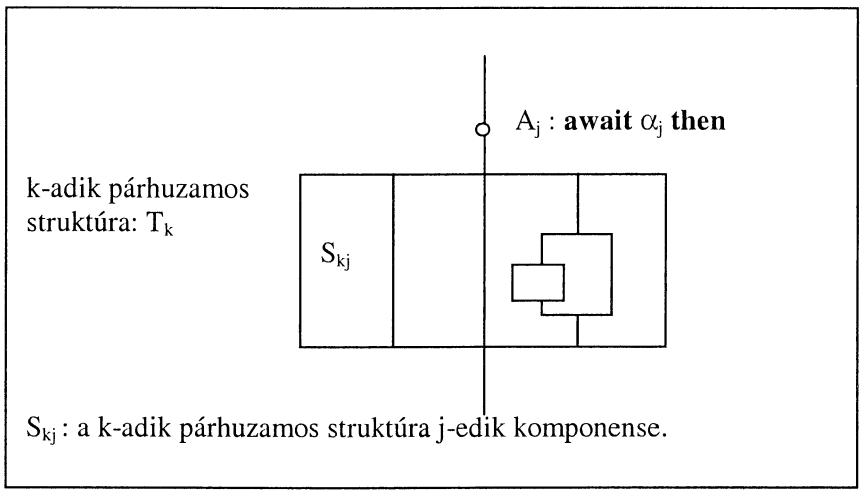
TELJES HELYESSÉG BIZONYÍTÁSA Bizonyítsuk be először, hogy a program komponensei helyesek gyenge teljes helyességi értelemben. Ezután bizonyítsuk be, hogy a párhuzamos program holtpontmentes.

HOLTPONTMENTESSÉG BIZONYÍTÁSA

- Vegyük számításba az összes olyan helyzetet, amelyben holtpont állhat elő.
- Mutassuk ki, hogy a holtpontállapot ténylegesen egyik esetben sem következhet be.

A bizonyításhoz vezessünk be jelöléseket. A nyelv szintaktikai definíciója alapján megállapíthatjuk, hogy egy párhuzamos program szekvenciális részekből és

az azok között elhelyezkedő párhuzamos részekből áll. A párhuzamos részek tetszőleges mértékig egymásba skatulyázhatók. Ezt szemlélteti a következő ábra. Ebben a szekvenciális részben kiemeltünk egy megvárakoztató utasítást.



Legyen megadva az S: {p} S_0; **parbegin** S_1 || ... || S_n **parend** {q} párhuzamos program.

a. Minden u_j , $j \in \{1, ..., n\}$, potenciálisan holtponthelyzetet eredményezhet S ben, ha
   - u_j egy A_i: **await** α_i ... megvárakoztató utasítás valamelyik S_i komponensben, vagy
   - u_j az üres utasítás az S_i befejeződése esetén.

b. Ha adott az S párhuzamos program $\{p_j\}$ S_j^* $\{ q_j \}$, gyenge teljes helyességi bizonyítása minden $j \in \{1, ..., n\}$ esetén, akkor a potenciális holtponthelyzetek állapotainak invariánsai:
   - A_i : **await** α_i ... esetén $pre(A_i) \wedge \alpha_i$;
   - Az S_j befejeződése esetén $post(S_j)$.

5.1. TÉTEL A holtpontmentesség egy elégséges feltétele. Legyen adott a {p} előfeltétellel az S: S_0; **parbegin** $S_1 \parallel S_2 \parallel \ldots \parallel S_n$ **parend** program.

Legyen

$$D(S): [(pre(A_1(x, y)) \wedge \sim\alpha_1(x, y)) \vee \ldots \vee$$

$$(pre(A_j(x, y)) \wedge \sim\alpha_j(x, y)) \vee \ldots] \vee [D_1(T_1) \vee \ldots \vee D_1(T_k) \vee \ldots],$$

ahol

$$D_1(T_k): [(post(S_{k1}) \vee D(S_{k1})) \wedge \ldots \wedge (post(S_{kj}) \vee D(S_{kj})) \wedge \ldots] \wedge$$

$$[D(S_{k1}) \vee \ldots \vee D(S_{kj}) \vee \ldots].$$

Ha D(S) = "false", akkor az S párhuzamos program holtpontmentes.

BIZONYÍTÁS A bizonyítást indirekt módon végezzük el az egymásba ágyazott **parbegin ... parend** utasítások számára vonatkozó indukcióval. Ehhez megvizsgáljuk először a lehetséges holtponthelyzeteket, majd megmutatjuk, hogy ezekben az esetekben a D(S) = "true".

Számbavétel:

- Holtponthelyzet áll elő, ha egy szekvenciális programban valahol blokkolt állapot következik be.

  Blokkolt az állapot, ha

  $$\{pre(A_j(x, y))\}\ \textbf{await}\ \alpha_j(x, y)\ \textbf{then}\ S_j(x, y)\ \textbf{ta};$$

  esetén

  $$pre(A_j(x, y)) \wedge \sim\alpha_j(x, y) = \text{"true"}.$$

- Valahol a szekvenciális részben blokkolt állapot következik be:

  $$(\exists j)(pre(A_j(x, y)) \wedge \sim\alpha_j(x, y)) = \text{"true"}),$$

  másként

  $$[(preA_1(x, y) \wedge \sim\alpha_1(x, y)) \vee \ldots \vee pre(A_j(x, y)) \wedge \sim\alpha_j(x, y)) \vee \ldots].$$

- Holtpont áll elő, ha a párhuzamos programon belül valamelyik párhuzamos részben holtpont következik be. Ekkor az indukció alapján nyerjük:

$$(\exists k)(D_1(T_k)) = \text{"true"}),$$

másként

$$[D_1(T_1) \vee \ldots \vee D_1(T_k) \vee \ldots].$$

Így kapjuk a holtpontállapotra, hogy

$$D(S):[(pre(A_1(x, y)) \wedge \sim\alpha_1(x, y)) \vee \ldots \vee$$
$$(pre(A_j(x, y)) \wedge \sim\alpha_j(x, y)) \vee \ldots] \vee [D_1(T_1) \vee \ldots \vee D_1(T_k) \vee \ldots].$$

Végül rekurzióval kapjuk $D_1(T_k)$ értékét. Ugyanis a párhuzamos részek esetén T_k-ban mindegyik S_{kj} vagy befejeződött, vagy holtpontra jutott, ugyanakkor van azonban olyan, amelyik holtpontra jutott. Így kapjuk, hogy

$$D_1(T_k): [(post(S_{k1}) \vee D(S_{k1})) \wedge \ldots \wedge (post(S_{kj}) \vee D(S_{kj})) \wedge \ldots] \wedge$$
$$[D(S_{k1}) \vee \ldots \vee D(S_{kj}) \vee \ldots].$$

Az összes lehetséges holtponthelyzetet megvizsgáltuk, minden esetben kimutattuk, hogy blokkolt állapot esetén D(S) = "true", ha tehát az S program esetében be tudjuk bizonyítani, hogy D(S) = "false", akkor az S párhuzamos program holtpontmentes.

Egy párhuzamos program **teljesen helyes**, ha:

a $\{p_i\}\ S_i^*\ \{q_i\}$, $i \in \{1, \ldots, n\}$ gyenge helyességi tételek igazak **és**

interferenciamentesek és

$(p \Rightarrow p_1 \wedge \ldots \wedge p_n)$ **és**

$q_1 \wedge \ldots \wedge q_n \Rightarrow q$ **és** a

parbegin $S_1 \| \ldots \| S_n$ **parend** program **holtpontmentes**

$\{p\}$ **parbegin** $S_1 \| \ldots \| S_n$ **parend** $\{q\}$

5.2. PÉLDA Legyen a párhuzamos program a következő:

$$S = T: \textbf{parbegin } S_1 \parallel S_2 \textbf{ parend};$$

ahol

$$S_1: S_{11}; \textbf{ await } \alpha_1 \textbf{ then } S_1' \textbf{ ta}; S_{12};$$

$$S_2: S_{21}; \textbf{ await } \alpha_2 \textbf{ then } S_2' \textbf{ ta}; S_{22}.$$

S_{11}, S_{12}, S_{21}, S_{22} determinisztikus programok, amelyek nem tartalmaznak megvárakoztató utasítást. Írjuk fel a D(S) értékét.

$$D(S): D_1(T);$$

$$D_1(T): [(post(S_1) \vee D(S_1)) \wedge (post(S_2) \vee D(S_2))] \wedge [D(S_1) \vee D(S_2)];$$

ahol

$$D(S_1): pre(A_1) \wedge \sim\alpha_1; \text{ azaz } S_1 \text{ várakozik.}$$

Legyen v_1: $pre(A_1) \wedge \sim\alpha_1$;

$$D(S_2): pre(A_2) \wedge \sim\alpha_2;$$

és legyen v_2: $pre(A_2) \wedge \sim\alpha_2$;

További jelölések:

$post(S_1)$: S_1 befejeződött. Jelölés: b_1: $post(S_1)$

$post(S_2)$: S_2 befejeződött. Jelölés: b_2: $post(S_2)$

$D(S)$: $[(b_1 \vee v_1) \wedge (b_2 \vee v_2)] \wedge [v_1 \vee v_2]$,

diszjunktív normál formára hozva kapjuk a számunkra kézzelfogható eredményt:

$$D(S): (v_1 \wedge b_2) \vee (b_1 \wedge v_2) \vee (v_1 \wedge v_2).$$

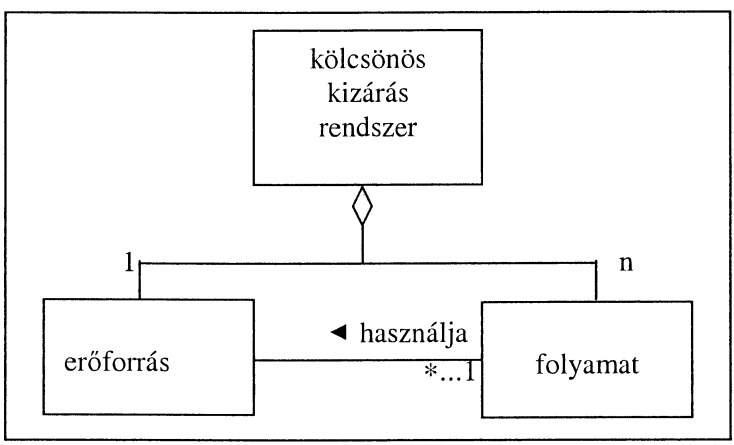
5.2. TÉTEL Legyen S párhuzamos program. Legyen A azon segédváltozóknak a halmaza, amelyek segítségével az $S^{\#}$ program holtpontmentességét $pre(S^{\#})$ mellett bebizonyítottuk. Ha az összes $s \in A$ segédváltozóra vonatkozó értékadó utasítást az $S^{\#}$ programból kitörölve az S programot kapjuk és $pre(S^{\#}) \Rightarrow pre(S)$, akkor S is holtpontmentes pre(S) mellett.

6. A KÖLCSÖNÖS KIZÁRÁS PROBLÉMÁJA

Kölcsönös kizárás esetén a folyamatok szinkronizálása a következőket jelenti:

1. Egy adott időpontban legfeljebb egy folyamat használhatja az erőforrást.
2. A rendszer legyen holtpontmentes.

A kölcsönös kizárás általános modelljét a következő ábra szemlélteti:



Adott egy erőforrás, amelyet n (>0)folyamat használ feladatainak megoldása során. Tegyük fel, hogy az erőforrást egyszerre csak egy folyamat használhatja. Legyen a párhuzamos rendszer a következő:

S: inicializálás; **parbegin** S_1 || ... || S_n **parend**;

ahol a folyamatok szerkezete:

S_i: **while** "true" **do**
NK_i: nem kritikus szakasz;
EI_i: erőforrás igénylése;
EH_i: erőforrás használata;
EF_i: erőforrás felszabadítása;
od;

ahol

$$(\forall i, j, 1 \leq i, j \leq n \wedge i \neq j)$$
$$[(\text{változó}(NK_i) \cup \text{változó}(EH_i))] \cap$$
$$[(\text{változó}(EI_j) \cup \text{változó}(EF_j))] = \{\};$$

NK_i és EH_i nem tartalmaz megvárakoztató utasítást.

Triviális megoldás a kölcsönös kizárásra:

S_j: **while** "true" **do**
NK_j;
await "true" **then** EH_j **ta**;
od;

6.1. ÁLLÍTÁS Legyenek megadva a {"true"} inicializálás {p}, $p \Rightarrow p_i$, $\{p_i\}$ S_i^* {"false"}, $i \in \{1, \ldots, n\}$, tételek. Ha ezek a tételek interferenciamentesek és $(i, j \in \{1, \ldots, n\}) \wedge i \neq j)(\text{pre}(EH_i) \wedge \text{pre}(EH_j) = \text{"false"})$, akkor teljesül a kölcsönös kizárás.

BIZONYÍTÁS Állításunk szerint egyszerre egynél több folyamat esetében nem teljesülhet az erőforrás használatának az előfeltétele, azaz egynél több folyamat nem férhet hozzá az erőforráshoz.

6.2. ÁLLÍTÁS Legyen S párhuzamos program. Legyen A azon segédváltozóknak a halmaza, amelyek segítségével az S* program esetén pre(S*) mellett a kölcsönös kizárást bebizonyítottuk. Ha az összes $s \in A$ segédváltozóra vonatkozó értékadó utasítást az S* programból kitörölve az S programot kapjuk, és pre(S*) $\Rightarrow$ pre(S), akkor S esetében is teljesül a kölcsönös kizárás pre(S) mellett.

MEGJEGYZÉS A mi esetünkben most holtpont akkor állhat elő, ha mindegyik párhuzamos komponens az erőforrásra vár:

$$D(S): pre(EI_1) \wedge \ldots \wedge pre(EI_n),$$

Ezért a holtpontmentesség feltétele:

$$pre(EI_1) \wedge \ldots \wedge pre(EI_n) = \text{"false"}.$$

6.1. PÉLDA Bizonyítsuk be, hogy a Peterson-féle algoritmus [BUR93] teljesíti a kölcsönös kizárási feltételt és holtpontmentes megoldást ad.

Adott a következő párhuzamos program:

$s \leftarrow 1$; $y_1 \leftarrow 0$; $y_2 \leftarrow 0$; **parbegin** $S_1 \parallel S_2$ **parend**;

S_1:	S_2:
while "true" **do**	**while** "true" **do**
nem kritikus szakasz$_1$;	nem kritikus szakasz$_2$;
$y_1 \leftarrow 1$;	$y_2 \leftarrow 1$;
$s \leftarrow 1$;	$s \leftarrow 2$;
wait $(y_2 = 0) \vee (s \neq 1)$;	**wait** $(y_1 = 0) \vee (s \neq 2)$;
kritikus szakasz$_1$;	kritikus szakasz$_2$;
$y_1 \leftarrow 0$; **od**;	$y_2 \leftarrow 0$; **od**;

Készítsük el a párhuzamos program invariánsát.

$$I(s, y_1, y_2): (s = 1 \vee s = 2) \wedge (y_1 = 0 \vee y_1 = 1) \wedge (y_2 = 0 \vee y_2 = 1)$$

Helyezzük el a program megfelelő pontjaiban az invariáns állításokat.

$s \leftarrow 1$; $y_1 \leftarrow 0$; $y_2 \leftarrow 0$; $\{I(s, y_1, y_2)\}$ **parbegin** $S_1 \parallel S_2$ **parend**;

S_1:	S_2:
$\{I(s, y_1, y_2) \wedge y_1 = 0\}$	$\{I(s, y_1, y_2) \wedge y_2 = 0\}$
while "true" **do**	**while** "true" **do**
nem kritikus szakasz$_1$;	nem kritikus szakasz$_2$;
$\{I(s, y_1, y_2) \wedge y_1 = 0\}$	$\{I(s, y_1, y_2) \wedge y_2 = 0\}$
$y_1 \leftarrow 1$;	$y_2 \leftarrow 1$;
$\{I(s, y_1, y_2) \wedge y_1 = 1\}$	$\{I(s, y_1, y_2) \wedge y_2 = 1\}$
$s \leftarrow 1$;	$s \leftarrow 2$;
$\{I(s, y_1, y_2) \wedge y_1 = 1\}$	$\{I(s, y_1, y_2) \wedge y_2 = 1\}$
wait $(y_2 = 0) \vee (s \neq 1)$;	**wait** $(y_1 = 0) \vee (s \neq 2)$;
kritikus szakasz$_1$;	kritikus szakasz$_2$;
$y_1 \leftarrow 0$; **od**;	$y_2 \leftarrow 0$; **od**;

A komponensekben elhelyezett állítások nyilvánvalóan igaz állítások, ha a komponenseket önmagukban tekintjük. Az interferenciamentesség is teljesül, mivel egyedüli közös változónk az s, amelynek az értéke csak olyan módon változik meg, hogy az invariáns változatlan marad.

A kölcsönös kizárás bizonyításához szükséges invariánsok:

$$\text{pre}(EH_1):$$

$$[(s = 1 \vee s = 2) \wedge (y_1 = 0 \vee y_1 = 1) \wedge (y_2 = 0 \vee y_2 = 1) \wedge y_1 = 1] \wedge$$

$$[(y_2 = 0) \vee (s \neq 1)];$$

$$\text{pre}(EH_2):$$

$$[(s = 1 \vee s = 2) \wedge (y_1 = 0 \vee y_1 = 1) \wedge (y_2 = 0 \vee y_2 = 1) \wedge y_2 = 1] \wedge$$

$$[(y_1 = 0) \vee (s \neq 2)];$$

A kölcsönös kizárás feltétele a mi esetünkben az, hogy

$$\text{pre}(EH_1) \wedge \text{pre}(EH_2) = \text{"false"},$$

ami tehát a következő:

$$[(s = 1 \vee s = 2) \wedge (y_1 = 0 \vee y_1 = 1) \wedge (y_2 = 0 \vee y_2 = 1) \wedge y_1 = 1] \wedge$$

$$[(y_2 = 0) \vee (s \neq 1)] \wedge$$

$$[(s = 1 \vee s = 2) \wedge (y_1 = 0 \vee y_1 = 1) \wedge (y_2 = 0 \vee y_2 = 1) \wedge y_2 = 1] \wedge$$

$$[(y_1 = 0) \vee (s \neq 2)] \Rightarrow$$

$$(s = 1 \vee s = 2) \wedge (s \neq 1) \vee (s \neq 2) = \text{"false"};$$

A holtpontmentesség feltétele a következő: Nem lehetséges az, hogy mindkét komponens a megvárakoztató utasítás őrfeltételének a teljesülésére vár. Formálisan ezt az állapotot a következő állítás írja le:

$$[(s = 1 \vee s = 2) \wedge (y_1 = 0 \vee y_1 = 1) \wedge (y_2 = 0 \vee y_2 = 1) \wedge y_1 = 1] \wedge$$

$$\sim[(y_2 = 0) \vee (s \neq 1)] \wedge$$

$$[(s = 1 \vee s = 2) \wedge (y_1 = 0 \vee y_1 = 1) \wedge (y_2 = 0 \vee y_2 = 1) \wedge y_2 = 1] \wedge$$

$$\sim[(y_1 = 0) \vee (s \neq 2)],$$

ami az előzővel megegyező módon "false" eredményre vezet.

6.2. PÉLDA A kölcsönös kizárás Dijkstra-féle megoldásának elemzése. Legyen a párhuzamos rendszer a következő:

S: $s \leftarrow 1$; **parbegin** $S_1 \parallel \ldots \parallel S_n$ **parend**;

ahol a folyamatok szerkezete:

```
Si: while "true" do
        NKi;
        P(s);
        EHi;
        V(s);
            od;
```

Alkalmazzuk a szemaforutasítások egyszerű implementációját megvárakoztató utasításokkal:

```
Si: while "true" do
        NKi;
    await s > 0 then s ← 0 ta;
        EHi;
    await "true" then s ← 1ta;
            od;
```

Vezessünk be egy segédváltozót, amelynek értéke 1, ha az S_i használja az erőforrást, különben pedig 0.

```
S': s ← 1; I1 ← 0; ... ; In ← 0; parbegin S1' || ... || Sn' parend;
```

ahol a folyamatok szerkezete:

S_i': **while** "true" **do**

NK_i;

await $s > 0$ **then** $s \leftarrow 0$; $I_i \leftarrow 1$; **ta**;

EH_i;

await "true" **then** $s \leftarrow 1$; $I_i \leftarrow 0$; **ta**;

od;

Invariáns:

$$\text{INV: } (s = 1 - (I_1 + \ldots + I_n)) \wedge (\forall i,\ 1 \leq i \leq n)(I_i = 0 \vee I_i = 1).$$

Helyezzük el a programban az állításokat.

$\{I_i = 0 \wedge \text{INV}\}$

$S_i'^*$: **while** "true" **do**

$\{I_i = 0 \wedge \text{INV}\}$

NK_i;

$\{I_i = 0 \wedge \text{INV}\}$

await $s > 0$ **then** $s \leftarrow 0$; $I_i \leftarrow 1$; **ta**;

$\{I_i = 1 \wedge \text{INV}\}$

EH_i;

$\{I_i = 1 \wedge \text{INV}\}$

await "true" **then** $s \leftarrow 1$; $k_i \leftarrow 0$; **ta**;

$\{I_i = 0 \wedge \text{INV}\}$

od;

Az állítások helyességének az ellenőrzése triviális feladat.

Az interferenciamentességhez azt kell ellenőrizni, hogy a globális változókra vonatkozó $I_i \neq a \wedge INV$ állításokat megváltoztathatja-e a vele párhuzamosan futó folyamat

await $s > 0$ **then** $s \leftarrow 0$; $I_i \leftarrow 1$; **ta**;

illetve **await** "true" **then** $s \leftarrow 1$; $I_i \leftarrow 0$; **ta**;

utasítása a szükséges előfeltételek mellett.

A válasz nyilvánvalóan az, hogy nem.

A **kölcsönös kizárás** bizonyításához kapjuk, hogy

$$(i, j \in \{1, \ldots, n\}) \wedge i \neq j)(pre(EH_i) \wedge pre(EH_j) = \text{"false"}.$$

Ugyanis

$$(i, j \in \{1, \ldots, n\}) \wedge i \neq j)((I_i = 1 \wedge INV \wedge s = 1) \wedge (I_j = 1 \wedge INV \wedge s = 1) = \text{"false"}.$$

A **holtpontmentesség** bizonyítása:

$$D(S): pre(EI_1) \wedge \ldots \wedge pre(EI_n) =$$

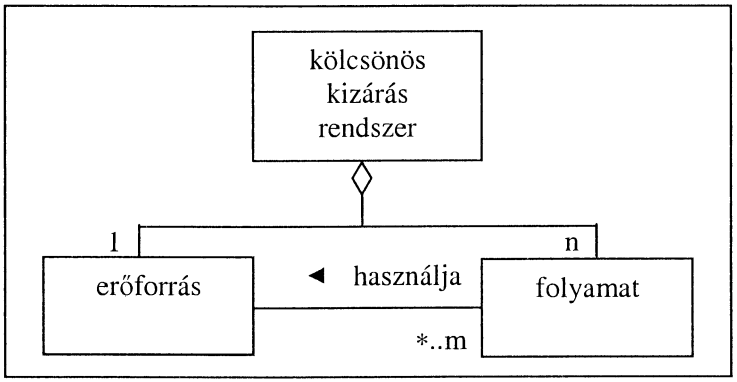
$$[(I_1 = 0 \wedge INV) \wedge \ldots \wedge (I_n = 0 \wedge INV)] \wedge s \neq 1 = \text{"false"},$$

mivel

$$(\forall i, 1 \leq i \leq n)(I_i = 0) \wedge INV \Rightarrow s = 1.$$

A segédváltozók elhagyásával az eredeti programra is érvényes a kölcsönös kizárás és a holtpontmentesség.

6.3. PÉLDA A kölcsönös kizárás modelljének általánosítása. Adva van egy erőforrás, amelyet n folyamat használ feladatainak megoldása során. Tegyük fel azonban, hogy az erőforrást egyszerre legfeljebb "m" folyamat használhatja, ahol $0 < m < n$. A probléma megoldásának osztályszerkezete a következő:



A modell megvalósításának vázlata a következő:

w ← m; **parbegin** $proc_1$ || ... || $proc_n$ **parbegin**;

$proc_j$:
while „true" **do**
nem kritikus $szakasz_j$;
P(w);
kritikus $szakasz_j$;
V(w);
nem kritikus $szakasz_j$;
od;
j = 1, ..., n.

Bizonyítsuk be, hogy az erőforrásra legfeljebb n - m folyamat várakozhat egyidejűleg.

BIZONYÍTÁS Ehhez segédváltozók kellenek, amelyek mutatják, hogy melyik folyamat használja az erőforrást. Legyenek ezek I_1, ..., I_n. Ekkor a program:

$I_1 \leftarrow 0; \ldots; I_n \leftarrow 0;$

parbegin ... || **while** "true" **do**

nem kritikus szakasz;

await $w > 0$ **then** $w \leftarrow w - 1; I_j \leftarrow 1$ **ta**;

kritikus szakasz;

await "true" **then** $w \leftarrow w + 1; I_j \leftarrow 1$ **ta**;

nem kritikus szakasz;

od; || ... **parend**;

Írjuk fel az invariánst.

INV: $(w = m - (I_1 + \ldots + I_n)) \wedge (0 < m < n) \wedge (\forall j, 1 \leq j \leq n)(I_j = 0 \vee I_j = 1)$.

Készítsük el a program annotált formáját.

$I_1 \leftarrow 0; \ldots; I_n \leftarrow 0;$

{INV}

parbegin ... || **while** "true" **do**

nem kritikus szakasz;

$\{INV \wedge I_j = 0\}$

await $w > 0$ **then** $w \leftarrow w - 1; I_j \leftarrow 1$ **ta**;

$\{INV \wedge I_j = 1\}$

kritikus szakasz;

await "true" **then** $w \leftarrow w + 1; I_j \leftarrow 1$ **ta**;

$\{INV \wedge I_j = 0\}$

nem kritikus szakasz;

od; || ... **parend**;

Tegyük fel, hogy az erőforrásra k folyamat várakozik, és $k > n - m$, ami tehát pozitív szám. Ekkor k darab I érték 0, és az invariánsunk a következő:

$$(w = m - (I_1 + \ldots + I_n)) \wedge (0 < m < n) \wedge (I_1 = 0 \wedge \ldots \wedge I_k = 0) \wedge$$

$$(I_{k+1} = 1 \wedge \ldots \wedge I_n = 1) \wedge (k > n - m) \wedge (w \leq 0).$$

Ebből kapjuk, hogy

$(w = m - (n - k) \wedge (w \leq 0) \Rightarrow m - n + k \leq 0$, azaz $k \leq n - m$, ami ellentmondás.

7. A KIÉHEZTETÉS

Párhuzamos rendszerekben az egymással párhuzamosan futó folyamatok versenyeznek egymással az erőforrások használatáért. Hibaként előfordulhat ezekben a rendszerekben az, hogy a közös erőforrást a folyamatok egy csoportja monopolizálni tudja, azaz végtelen hosszú ideig lekötve tartja.

DEFINÍCIÓ Legyen adva az

$$S: \textbf{parbegin } S_1 \parallel \ldots \parallel S_n \textbf{ parend}$$

párhuzamos rendszer, ahol az S_j, ($j = 1, 2, \ldots, n$) programok ciklikus folyamatok:

$$S_j : \textbf{while } \text{"true"} \textbf{ do } S_j' \textbf{ od};$$

Azt mondjuk, hogy a rendszer nem mentes a kiéheztetetéstől, ha a párhuzamosan futó folyamatok egy csoportja monopolizálhatja a párhuzamos rendszer közös erőforrását.

Az erőforrás monopolizálása azt jelenti, hogy a párhuzamosan futó folyamatok egy nem üres csoportjában mindig van legalább egy olyan folyamat, amely lekötve tartja az erőforrást, miközben a velük párhuzamos folyamatok nem üres csoportjának tagjai folyamatosan várakoznak arra.

A RENDSZER KIÉHEZTETÉSMENTESSÉGÉNEK BIZONYÍTÁSA

- Vegyük számításba a rendszerben azokat a helyzeteket, amelyek kiéheztetés szempontjából potenciálisan számításba jönnek.

- Mutassuk ki, hogy ezek közül egy sem fordulhat elő.

DEFINÍCIÓ Az (R_1, ... , R_n) utasításoknak az n-esét kiéheztetés szempontjából potenciálisan veszélyesnek nevezzük, ha a következő feltételeknek tesznek eleget:

- Az (R_1, ..., R_n) utasításoknak egy nem üres részhalmaza olyan **await** megvárakoztató utasítás, amelynél a folyamatok egy olyan csoportja várakozhat a közös erőforrás használatára, amelyet egy másik csoport folyamatai kizárhatnak az erőforrás használatából. Legyen ezeknek a száma k, $0 < k < n$;
- Az (R_1, ..., R_n) utasításoknak egy nem üres részhalmaza olyan **await** megvárakoztató utasítás, amelynél a folyamatok egy csoportja számára folyamatosan biztosítva van a belépési feltétel teljesülése, akkor is, ha az erőforrást a csoport más tagjai használják; vagy olyan utasítás, amely az erőforrás használatát jelenti. Jelölje ezeknek a számát j, $0 < j \leq n - k$;
- Az (R_1, ... , R_n) utasításoknak egy részhalmaza pedig olyan üres utasítás, amelynél valamelyik S_i végrehajtása befejeződött. Ezeknek a száma $n - j - k$.

DEFINÍCIÓ Legyen adva az S program teljes helyességének $\{p_i\}\ S_i^*\ \{q_i\}$ bizonyítása.

- Az első csoportba tartozó utasítások előfeltételei a $pre(A_j) \wedge \sim\alpha_j$, ha A_j: **await** α_i ..., és a csoportban várakozók együttes blokkolt állapotának feltétele:

$$(pre(A_1) \wedge \sim\alpha_1 \wedge \ldots \wedge (pre(A_k) \wedge \sim\alpha_k);\ \text{ahol}\ 0 < k < n;$$

- A második csoportba tartozó megvárakoztató utasítások belépési feltételei mind teljesülnek, ezt az állapotot a következő állítás írja le:

$$(pre(B_1) \wedge \beta_1 \wedge \ldots \wedge (pre(B_j) \wedge \beta_j);\ \text{ahol}\ 0 < j \leq n - k;$$

- Végül a befejezett folyamatok együttes állapotát leíró invariáns:

$$post(S_1) \wedge \ldots \wedge post(S_i), \text{ ahol } i = n - k - j.$$

7.1. TÉTEL Ha

$$[(pre(A_1) \wedge \sim\alpha_1 \wedge \ldots \wedge (pre(A_k) \wedge \sim\alpha_k)] \wedge$$
$$[(pre(B_1) \wedge \beta_1 \wedge \ldots \wedge (pre(B_j) \wedge \beta_j)] \wedge$$
$$[post(S_1) \wedge \ldots \wedge post(S_i)] = \text{"false"},$$

akkor a párhuzamos rendszer kiéheztetésmentes.

Bizonyítás. A definíció alapján az állítás nyilvánvaló.

7.2. PÉLDA Készítsük el a következő informális specifikációnak megfelelő leegyszerűsített adatbázisrendszert.

- Az adatbázist $n > 1$ darab felújító és $m > 1$ darab lekérdező folyamat használja.
- A felújító folyamatok egymást kölcsönösen kizárják az adatbázis használatából.
- A lekérdező folyamatok közül tetszőleges számú folyamat használhatja egyidejűleg az adatbázist.
- A felújító folyamat és a lekérdező folyamat egymást kölcsönösen kizárják az adatbázis használatából.

MEGOLDÁS

kezdőértékek beállítása;
parbegin $writer_1$; || ... || $writer_n$; || $reader_1$; || ... || $reader_m$ **parend**;

A felújító folyamatok egymást kölcsönösen kizárják az adatbázis használatából. A kölcsönös kizárást a w szemafor segítségével valósíthatjuk meg, amelynek kezdőértéke 1.

writer$_j$,
$j = 1, \ldots, n$:

```
while "true" do
        információgyűjtés;
            P(w);
        adatbázis-felújítás;
            V(w);
    od;
```

A lekérdező folyamatok közül tetszőleges számú folyamat használhatja egyidejűleg az adatbázist, de egy felújító folyamat és egy lekérdező folyamat egymást kölcsönösen kizárja. Ehhez vezessünk be a lekérdező folyamatok részére egy számlálót, amely azt mutatja, hogy hányadik lekérdező folyamat kíván belépni az adatbázis használatába. Az első állítsa tilosra a w szemafort a felújító folyamatok előtt, az utolsó pedig állítsa vissza a szemafort szabad jelzésre. A számláló legyen rcnt, amelynek a kezdőértéke 0, és a számlálás műveleteinek kölcsönös kizárására vezessük be az sr szemafort, amelynek kezdőértéke 1.

reader$_j$,
$j = 1, \ldots, m$:

```
while "true" do
        P(sr);   rcnt ← rcnt + 1; if rcnt = 1 then P(w) fi;
        V(sr);   az adatbázis lekérdezése;
        P(sr);   rcnt ← rcnt - 1; if rcnt = 0 then V(w) fi;
        V(sr);   a válasz kiértékelése;
    od;
```

Végezzük el a modell tulajdonságainak elemzését. Először a P(s), V(s) utasításokat **await** utasításokkal valósítsuk meg. Annak nyilvántartására, hogy melyik felújító használja az adatbázist, vezessük be az $I_1, \ldots, I_n$ segédváltozókat, amelyek kezdőértéke 0. A felújító folyamatokban az implementáció:

$P_w(w)$: **await** $w > 0$ **then** $w \leftarrow w - 1$; $I_j \leftarrow 1$ **ta**;

$V_w(w)$: **await** "true" **then** $w \leftarrow w + 1$; $I_j \leftarrow 0$ **ta**;

Annak nyilvántartására, hogy lekérdező használja-e az adatbázist, vezessük be az R segédváltozót, amelynek a kezdőértéke 0.

$P_r(w)$: **await** $w > 0$ **then** $w \leftarrow w - 1$; $R \leftarrow 1$ **ta**;

$V_r(w)$: **await** "true" **then** $w \leftarrow w + 1$; $R \leftarrow 0$ **ta**;

A számlálást védő szemaforra vonatkozó műveletek megvalósítása legyen a következő:

P(sr) : **await** $sr > 0$ **then** $sr \leftarrow sr - 1$ **ta**;

V(sr) : **await** "true" **then** $sr \leftarrow sr + 1$**ta**;

Itt most nem vezetünk be segédváltozót az elemzéshez, mert a számlálás kölcsönös kizárására felhasználjuk a Dijkstra-féle megoldás során elvégzett vizsgálatunk eredményét.

A program:

$w \leftarrow 1$; $I_1 \leftarrow 0$; ... $I_n \leftarrow 0$; $sr \leftarrow 1$; $rcnt \leftarrow 0$; $R \leftarrow 0$;

parbegin $writer_1$; || ... || $writer_n$; || $reader_1$; || ... || $reader_m$ **parend**;

$writer_j$,

$j = 1, \ldots, n$;

while "true" **do**

információ gyűjtése;

await $w > 0$ **then** $w \leftarrow w - 1$; $I_j \leftarrow 1$ **ta**;

az adatbázis felújítása;

await "true" **then** $w \leftarrow w + 1$; $I_j \leftarrow 0$ **ta**;

od;

$reader_j$,
$j = 1, \ldots, m$;

```
while "true" do
        P(sr);   rcnt ← rcnt + 1;
                 if rcnt = 1 then
        await w > 0 then w ← w - 1; R ← 1 ta;
                 fi;
        V(sr);   az adatbázis lekérdezése;
        P(sr);   rcnt ← rcnt - 1;
                 if rcnt = 0 then
        await "true" then w ← w + 1; R ← 0 ta;
                 fi;
        V(sr);   a válasz kiértékelése;
             od;
```

Szerkesszük meg a rendszer invariánsát:

$$\text{INV: } w = (1 - (I_1 + \ldots + I_n) - R) \wedge (\forall j, 1 \leq j \leq n)(I_j = 0 \vee I_j = 1) \wedge$$

$$(R = 0 \vee R = 1) \wedge sr \leq 1 \wedge 0 \leq rcnt;$$

Nyilvánvaló, hogy a kezdőértékek beállítása után az invariáns érvényes lesz:

$$w \leftarrow 1; I_1 \leftarrow 0; \ldots I_n \leftarrow 0; sr \leftarrow 1; rcnt \leftarrow 0; R \leftarrow 0;$$

$$\{\text{INV}\}$$

Készítsük el a párhuzamos komponensek annotációs változatait.

$\{INV \wedge R \geq 0 \wedge rcnt \geq 0\}$

while "true" **do**

$\{INV \wedge R \geq 0 \wedge rcnt \geq 0\}$

await $sr > 0$ **then** $sr \leftarrow sr - 1$ **ta**;

$\{INV \wedge R \geq 0 \wedge rcnt \geq 0 \wedge sr \leq 0\}$

$rcnt \leftarrow rcnt + 1$;

$\{INV \wedge R \geq 0 \wedge rcnt \geq 1 \wedge sr \leq 0\}$

if $rcnt = 1$ **then**

$\{INV \wedge R \geq 0 \wedge rcnt = 1 \wedge sr \leq 0\}$

await $w > 0$ **then** $w \leftarrow w - 1$; $R \leftarrow 1$ **ta**; **fi**;

await "true" **then** $sr \leftarrow sr + 1$**ta**;

$\{INV \wedge R = 1 \wedge rcnt \geq 1 \wedge sr \leq 1 \wedge w \leq 0\}$

$reader_j^*$: az adatbázis lekérdezése;

$\{INV \wedge R = 1 \wedge rcnt \geq 1 \wedge sr \leq 1 \wedge w \leq 0\}$

await $sr > 0$ **then** $sr \leftarrow sr - 1$ **ta**;

$\{INV \wedge R = 1 \wedge rcnt \geq 1 \wedge sr \leq 0 \wedge w \leq 0\}$

$rcnt \leftarrow rcnt - 1$;

$\{INV \wedge R = 1 \wedge rcnt \geq 0 \wedge sr \leq 0 \wedge w \leq 0\}$

if $rcnt = 0$ **then**

await "true" **then** $w \leftarrow w + 1$; $R \leftarrow 0$ **ta**; **fi**;

$\{INV \wedge (R = 0 \wedge rcnt = 0) \vee (R = 1 \wedge rcnt > 0) \wedge sr \leq 0 \wedge w \leq 1\}$

await "true" **then** $sr \leftarrow sr + 1$**ta**;

$\{INV \wedge R \geq 0 \wedge rcnt \geq 0\}$

a válasz kiértékelése;

od;

writer$_j^*$:

$\{INV \wedge I_j = 0\}$

while "true" **do**

információ gyűjtése;

$\{INV \wedge I_j = 0\}$

await $w > 0$ **then** $w \leftarrow w - 1; I_j \leftarrow 1$ **ta**;

$\{INV \wedge I_j = 1 \wedge w \leq 0\}$

az adatbázis felújítása;

$\{INV \wedge I_j = 1 \wedge w \leq 0\}$

await "true" **then** $w \leftarrow w + 1; I_j \leftarrow 0$ **ta**;

$\{INV \wedge I_j = 0\}$

od;

7.1. PÉLDA Végezzük el az adatbázismodell elemzését!

MEGOLDÁS Megválaszolandó kérdések:

- Igaz-e a gyenge teljes helyesség?
- Teljesülnek-e a programban elhelyezett állítások?
- A tételek interferenciamentesek-e?

Ezeket az egyszerű vizsgálatokat most mellőzük, foglalkozzunk inkább a párhuzamosság alábbi tulajdonságainak vizsgálatával:

1. A felújító folyamatok egymást kölcsönösen kizárják az adatbázis használatából.
2. A lekérdező folyamatok egymást kölcsönösen kizárják az "rcnt" számlálón történő művelet végzéséből.

Ezzel a két kérdéssel most nem kell foglalkoznunk, mivel itt a kölcsönös kizárás korábban vizsgált szerkezete jelenik meg a megoldásunkban. Az ott elvégzett bizonyítások alapján mondhatjuk, hogy a felsorolt két tulajdonság teljesül.

3. Ha felújító folyamat használja az adatbázist, akkor lekérdező folyamat nem férhet hozzá.

$$(INV \wedge I_j = 1 \wedge w \leq 0) \wedge (INV \wedge R \geq 0 \wedge rcnt = 1 \wedge sr \leq 0 \wedge w > 0) = \text{"false"}.$$

4. Ha lekérdező folyamat használja az adatbázist, akkor felújító folyamat nem férhet hozzá.

$$(INV \wedge R = 1 \wedge rcnt \geq 1 \wedge sr \leq 1 \wedge w \leq 0) \wedge (INV \wedge I_j = 0 \wedge w > 0) = \text{"false"}.$$

5. A rendszer holtpontmentes.

$$[(\forall j, 1 \leq j \leq n)(INV \wedge I_j = 0)] \wedge (INV \wedge R = 0 \wedge rcnt = 1 \wedge sr \leq 0) \wedge (w \leq 0) = \text{"false"}$$

6. A rendszer nem mentes a kiéheztetéstől.

   Lehet olyan konfiguráció, hogy minden felújító folyamat vár:

$$[INV \wedge (I_1 = 0 \wedge \ldots \wedge I_n = 0) \wedge w \leq 0];$$

   és az erőforrást csak lekérdező folyamatok használják:

$$[INV \wedge R = 1 \wedge rcnt \geq 1 \wedge sr \leq 1 \wedge w \leq 0];$$

   és további lekérdező folyamatok részére szabad az út a belépéshez:

$$[(INV \wedge R \geq 0 \wedge rcnt \geq 1 \wedge sr \leq 0) \wedge (rcnt \neq 1)];$$

   azaz

$$[INV \wedge (I_1 = 0 \wedge \ldots \wedge I_n = 0) \wedge w \leq 0] \wedge$$
$$[INV \wedge R = 1 \wedge rcnt \geq 1 \wedge sr \leq 1 \wedge w \leq 0] \wedge$$
$$[(INV \wedge R \geq 0 \wedge rcnt \geq 1 \wedge sr \leq 0) \wedge (rcnt \neq 1)] = \text{"true"}.$$

7.2. PÉLDA Legyen a felújító folyamatoknak elsőbbségük a lekérdező folyamatokkal szemben.

MEGOLDÁS

1. Az első belépni szándékozó felújító folyamat állítsa tilosra a szemafort a felújító folyamatok előtt. A felújító folyamatok tehát tartsák nyilván belé-

pési szándékukat. Erre a célra vezessük be a wcnt számlálót, amelynek kezdőértéke 0, és a számlálóval történő műveletek kölcsönös kizárására használjuk az sw szemafort, amelynek kezdőértéke 1.

2. A lekérdező folyamatok, amelyek használják az adatbázist – ha végeztek – , jöjjenek ki, de újabbak ne mehessenek be. Ezt megoldhatjuk úgy, hogy a folyamatok belépését kölcsönös kizárásos módon valósítjuk meg egy r szemafor bevezetésével, amelynek kezdőértéke 1.

A program, amelyben a felújító folyamatoknak elsőbbségük van a lekérdező folyamatokkal szemben:

```
w ← 1; I1 ← 0; ...; In ← 0; sr ← 1; sw ← 1; rcnt ← 0;
R ← 0; r ← 1; wcnt ← 0;
parbegin writer1; || ... || writern; || reader1; || ... || readerm parend;
```

writer$_j$, $j = 1, \ldots, n$

```
while "true" do
        információgyűjtés;
P(sw);  wcnt ← wcnt + 1;
        if wcnt = 1 then P(r) fi;
V(sw);
        P(w);
        adatbázis-felújítás;
        V(w);
P(sw);  wcnt ← wcnt - 1;
        if wcnt = 0 then V(r) fi;
V(sw);
                od;
```

reader$_j$,
$j = 1, \ldots, m$:

```
while "true" do
P(r);
        P(sr);
                rcnt ← rcnt + 1;
                if rcnt = 1 then
        await w > 0 then w ← w - 1; R ← 1 ta;
                fi;
        V(sr);
V(r);
        az adatbázis lekérdezése;
P(sr);
                rcnt ← rcnt - 1;
                if rcnt = 0 then
        await "true" then w ← w + 1; R ← 0 ta;
                fi;
V(sr);
        a válasz kiértékelése;
            od;
```

Sajnos, a lekérdező folyamatok belépésének kölcsönös kizárása nem elégséges a helyes megoldáshoz. A belépés vizsgálatánál ugyanis az újabb felújító folyamatok és a lekérdezők is az r szemafornál várakoznak. Nincs tehát garantálva az, hogy közülük a felújító folyamatok léphetnek be elsőnek. Ezt garantálhatjuk egy újabb szemafor bevezetésével, amelyet az r szemafor előtt helyezünk el, és a lekérdező folyamatokat ennél a szemafornál várakoztatjuk meg. Legyen ez a szemafor re, amelynek kezdőértéke 1. Így már a lekérdező folyamatok jó megoldása a következő:

$reader_j$,
$j = 1, \ldots, m$:

```
while "true" do
        P(re);
            P(r);
                P(sr);
                    rcnt ← rcnt + 1;
                    if rcnt = 1 then
                await w >0 then w ← w - 1; R ← 1 ta;
                    fi;
                V(sr);
            V(r);
        V(re);
                    az adatbázis lekérdezése;
                    P(sr);
                        rcnt ← rcnt - 1;
                        if rcnt = 0 then
                    await "true" then w ← w + 1; R ← 0 ta;
                        fi;
                    V(sr);
                    a válasz kiértékelése;
                od;
```

Sajnos, ez a rendszer sem mentes a kiéheztetéstől, csak most a felújító folyamatok monopolizálhatják az adatbázist.

7.3. PÉLDA A termelő–fogyasztó rendszer. A termelő–fogyasztó rendszer lényege az, hogy a termelő az áruit berakja egy raktárba, amelyet onnan a fogyasztó kiszállít. Legyen ez a raktár egy sor, amelybe legfeljebb n darab áru

helyezhető el. Ha van üres hely, a termelő mindig egy darab árut helyez el a raktárban, és ezt a sor végére teszi. A fogyasztó szintén egy darab árut vesz ki egyszerre a raktárból, és ezt a sor elejéről veszi el.

MEGOLDÁS A problémában megadott szinkronizáció megvalósítása céljából vezessünk be két paramétert:

empty: a sorban lévő üres elemek száma,

full: a foglalt elemek száma.

Jelölje y a sort. Legyenek annak műveletei:

add(y, x): az y sorban az x áru elhelyezése;

read(y): a sor elejéről egy áru kiemelése;

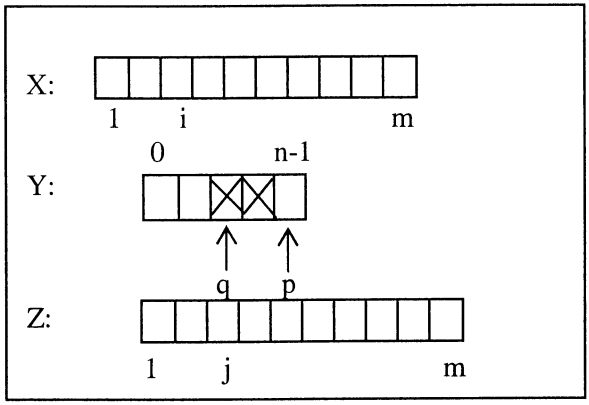
rem(y): a sor elejéről kiemelt áru után maradó raktár előállítása;

Kezdőértékek:

y = init, full = 0, empty = n.

termelő:	fogyasztó:
while "true" **do**	**while** "true" **do**
az x termék előállítása;	P(full);
P(empty);	(z, y) ← (read(y), rem(y));
y ← add(y, x);	V(empty);
V(full);	a z termék feldolgozása;
od;	**od**;

Ennek a problémának egy szokásos véges modellje a következő:



A termelő egy m hosszúságú vektorból olvassa ki egymás után az elemeket. Az x termék előállításának a következő művelet felel meg: $(x \leftarrow X[i]; i + 1)$. Ezt az x terméket a termelő elhelyezi egy n hosszúságú y vektorban, ahol $m > n > 0$. Ekkor a megfeleltetés az általános modell műveletével a következő:

$$y \leftarrow \text{add}(y, x): (y[p_{\text{mod}(n)}] \leftarrow x, p + 1).$$

A fogyasztó kiolvassa a terméket az y sorból. Ekkor a megfeleltetés:

$$(z, y) \leftarrow (\text{read}(y), \text{rem}(y)): (z \leftarrow Y[q_{\text{mod}(n)}]; q = q + 1)$$

A kiolvasott z terméket a fogyasztó elhelyezi egy m hosszúságú z sorban. A z termék feldolgozásának a művelete a mi esetünkben: $(Z[j] \leftarrow z; j + 1)$.

Figyelembe véve, hogy most

$$\text{full} = p - q;$$

$$\text{empty} = n - (p - q);$$

a program teljes helyességére vonatkozó tételünk a következő lesz:

$\{p = 0 \wedge q = 0 \wedge i = 1 \wedge j = 1 \wedge 0 < n \wedge 0 < m\}$

parbegin termelő || fogyasztó **parend**

$\{(\forall k, 1 \leq k \leq m)(z[k] = x[k])\}$

ahol

termelő:	fogyasztó:
while $i \leq m$ **do**	**while** $j \leq m$ **do**
$X \leftarrow X[i];$	**await** $p - q > 0$ **then ta**;
await $n > p - q$ **then ta**;	$z \leftarrow Y[q_{mod(n)}];$
$Y[p_{mod(n)}] \leftarrow x;$	$q = q + 1$
$p \leftarrow p + 1;$	$Z[j] \leftarrow z;$
$i + 1;$	$j + 1;$
od;	**od**;

Bizonyítsuk be, hogy a program a specifikáció szerint helyes.

Az invariáns a termelő esetében:

$$I: 1 \leq i \leq m + 1 \wedge 0 \leq p - q \leq n \wedge (\forall k, 0 \leq q < k \leq p \wedge$$
$$Y[(k \underline{\text{mod}}\ n) - 1] = X[k]) \wedge 1 \leq j \leq m + 1.$$

Az invariáns a fogyasztó esetében:

$$I \wedge IF, \text{ ahol } IF: (\forall k, 1 \leq k < j)(Z[k] = X[k]).$$

Nyilvánvaló, hogy a program indulásakor

$$q = 0 \wedge p = 0 \wedge i = 1 \wedge j = 1 \wedge 0 < n \wedge 0 < m \Rightarrow I \wedge IF.$$

Az annotációkkal kiegészített programvázlat ezért a következő lesz:

$\{full = 0 \wedge empty = m \wedge q = 0 \wedge p = 0 \wedge i = 1 \wedge j = 1 \wedge 0 < n \wedge 0 < m\}$

parbegin

$\{I \wedge i = p + 1\}$ termelő $\{I \wedge i = p + 1 \wedge i = m + 1\} \parallel$

$\{I \wedge IF \wedge j = q + 1\}$ fogyasztó $\{I \wedge IF \wedge j = q + 1 \wedge j = m + 1\}$

parend

$\{(\forall k, 1 \leq k \leq m)$

A termelő–fogyasztó rendszer teljes helyességének a bizonyításához a következőket kell belátni:

- A termelőre vonatkozó tétel önmagában teljesen helyes.
- A fogyasztóra vonatkozó tétel önmagában teljesen helyes.
- A termelőre és a fogyasztóra megfogalmazott helyességi tételeink interferenciamentesek.
- A rendszer gyenge értelemben teljesen helyes.
- A rendszer holtpontmentes.

Készítsük el a két komponens annotációs változatát.

termelő:

$\{I \wedge i = p + 1\}$

while $i \leq m$ **do**

$\{I \wedge i = p + 1 \wedge i \leq m\}$

$X \leftarrow X[i];$

$\{I \wedge i = p + 1 \wedge i \leq m \wedge x = X[i]\}$

await $n > p - q$ **then ta**;

$\{I \wedge i = p + 1 \wedge i \leq m \wedge x = X[i] \wedge (n > p - q)\}$

$Y[p_{mod(n)}] \leftarrow x;$

$\{I \wedge i = p + 1 \wedge i \leq m \wedge x = X[i] \wedge (n > p - q) \wedge Y[p_{mod(n)}] = X[i]\}$

$p \leftarrow p + 1;$

$\{I \wedge i = p \wedge i \leq m\}$

$i + 1;$

$\{I \wedge i = p + 1\}$

od;

$\{I \wedge i = p + 1 \wedge i \leq m + 1\}$

fogyasztó: $\{I \wedge IF \wedge j = q + 1\}$

while $j \leq m$ **do**

$\{I \wedge IF \wedge j = q + 1 \wedge j \leq m\}$

await $p - q > 0$ **then ta**;

$\{I \wedge IF \wedge j = q + 1 \wedge j \leq m \wedge (p - q > 0)\}$

$z \leftarrow Y[q_{mod(n)}]$;

$\{I \wedge IF \wedge j = q + 1 \wedge j \leq m \wedge (p - q > 0) \wedge z = X[j]\}$

$q = q + 1$

$\{I \wedge IF \wedge j = q \wedge j \leq m \wedge (p - q > 0) \wedge z = X[j]\}$

$Z[j] \leftarrow z$;

$\{I \wedge IF \wedge j = q \wedge j \leq m \wedge (p - q > 0) \wedge Z[j] = X[j]\}$

$j + 1$;

$\{I \wedge IF \wedge j = q + 1 \wedge j \leq m + 1\}$

od;

$\{I \wedge IF \wedge j = q + 1 \wedge j = m + 1\}$

A komponensek parciális helyessége: A komponensek parciális helyességének bizonyítása a programban elhelyezett állítások lépésről lépésre történőn ellenőrzését jelenti. Ennek alapján a komponensek önmagukban való parciális helyességét elfogadhatjuk.

Befejeződés: $I \Rightarrow (i \leq m + 1 \wedge j \leq m + 1)$ alapján nyilvánvaló a komponensek önmagukban való befejeződése.

Interferenciamentesség: A termelőnél csak az

$\{I \wedge i = p + 1 \wedge i \leq m \wedge x = X[i] \wedge (n > p - q) \wedge Y[p: \underline{mod}\ n] = X[i]\}$

$p \leftarrow p + 1$;

értékadás okozhatna problémát, de a fogyasztónál az invariánsban ezt a

$$p - q > 0$$

relációban használjuk fel, amit a fenti értékadás nyílván nem befolyásol.

Hasonlóan a fogyasztónál csak az

$$\{I \wedge IF \wedge j = q + 1 \wedge j \leq m \wedge (p - q > 0) \wedge z = X[j]\}$$

$$q = q + 1$$

okozhatna problémát, amit a termelőnél viszont az

$$n > p - q.$$

relációban használunk fel. Tételeink tehát interferenciamentesek.

A párhuzamos program parciális helyessége:

$$(I \wedge i = p + 1 \wedge i = m + 1) \wedge (I \wedge IF \wedge j = q + 1 \wedge j = m + 1) \Rightarrow$$

$$(\forall k, 1 \leq k \leq m)(Z[k] = x[k]),$$

ami figyelembe véve IF jelentését, $i = m + 1$ és $j = m + 1$ esetén nyilvánvalóan fennáll. A gyenge teljes helyesség bizonyítása ezzel befejeződött.

A holtpontmentesség bizonyítása: Vegyük számba a holtpontmentesség szempontjából kritikus eseteket, és adjuk meg ezek jellemző invariánsait.

v1: termelő várakozik: $i = p + 1 \wedge i \leq m \wedge p - q = n \wedge 0 < n \wedge 0 < m$;

v2: fogyasztó várakozik: $j = q + 1 \wedge j \leq m \wedge p - q = 0 \wedge 0 < n \wedge 0 < m$;

b1: termelő végére ért: $i = p + 1 \wedge i = m + 1 \wedge 0 < n \wedge 0 < m$;

b2: fogyasztó végére ért: $j = q + 1 \wedge j = m + 1 \wedge 0 < n \wedge 0 < m$;

Itt az invariánsoknak csak a bizonyításhoz szükséges részeit adtuk meg.

A holtpontmentességhez azt kell bizonyítanunk, hogy a következő esetek nem fordulhatnak elő:

- Nem lehetséges, hogy mindkettő várakozik:

$$v1 \wedge v2 \equiv (i = p + 1 \wedge i \leq m \wedge p - q = n \wedge j = q + 1 \wedge j \leq m \wedge p - q = 0) \wedge$$

$$(0 < n \wedge 0 < m) \equiv \text{"false"}$$

- Nem lehetséges, hogy a termelő várakozik, a fogyasztó befejezte futását:

$$v1 \wedge b2 \equiv (i = p + 1 \wedge i \leq m \wedge p - q = n) \wedge$$

$$(j = q + 1 \wedge j = m + 1 \wedge 0 < n \wedge 0 < m) \equiv \text{"false"}$$

- Nem lehetséges, hogy a fogyasztó várakozik, a termelő befejezte futását:

$$b1 \wedge v2 \equiv (j = q + 1 \wedge j \leq m \wedge p - q = 0) \wedge$$

$$(i = p + 1 \wedge i \leq m + 1 \wedge 0 < n \wedge 0 < m) \equiv \text{"false"}$$

Megmutattuk, hogy a program holtpontmentes. A teljes helyesség bizonyítása ezzel befejeződött. Itt most kiéheztetés nem fordulhat elő.

7.4. PÉLDA Bizonyítsuk be a következő program holtpontmentességét.

next ← 1; B ← "false";

parbegin S_1 || S_2 **parend**;

S_1:	S_2:
j ← 0;	i ← 0;
while B = "false" **do**	**while** B = "false" **do**
await next = 1	**await** next = 2
then next ← 2 **ta**;	**then** next ← 1 **ta**;
if X[j] = 0 **then** B ← "true" **fi**;	**if** Y[i] = 0 **then** B ← "true" **fi**;
j ← j + 1; **od**; next ← 2;	i ← i + 1; **od**; next ← 1;

BIZONYÍTÁS Szerkesszük meg a rendszer invariánsát.

$$I(i, j, next, Y, B): i \geq 0 \wedge j \geq 0 \wedge (next = 1 \vee next = 2) \wedge$$

$$(B = \text{"true"} \vee B = \text{"false"}) \wedge ((Y[i] = 0 \vee Y[j] = 0) \Rightarrow B = \text{"true"}));$$

Készítsük el a program annotációs formáját.

next ← 1; B ← "false"; {next = 1 ∧ B = "false"}

parbegin S_1 || S_2 **parend**;

S_1:	S_2:
j ← 0;	i ← 0;
{I(i, j, next, Y, B) ∧ j = 0}	{I(i, j, next, Y, B) ∧ i = 0}
while B = "false" **do**	**while** B = "false" **do**
{I(i, j, next, Y, B)}	{I(i, j, next, Y, B)}
await next = 1	**await** next = 2
then next ← 2 **ta**;	**then** next ← 1 **ta**;
if X[j] = 0 **then** B ← "true" **fi**;	**if** Y[i] = 0 **then** B ← "true" **fi**;
j ← j + 1; **od**; next ← 2;	i ← i + 1; **od**; next ← 1;
{I(i, j, next, Y, B) ∧ next = 2}	{I(i, j, next, Y, B) ∧ next = 1}

A holtpontmentesség feltétele:

- Nem lehetséges, hogy mindkettő vár:

  [i ≥ 0 ∧ j ≥ 0 ∧ (next = 1 ∨ next = 2) ∧ (B = "true" ∨ B = "false") ∧

  ((Y[i] = 0 ∨ Y[j] = 0) ⇒ B = "true") ∧ next ≠ 2 ∧ next ≠ 1] = "false".

- Nem lehetséges, hogy S_1 vár és S_2 futását befejezte:

  [i ≥ 0 ∧ j ≥ 0 ∧ (next = 1 ∨ next = 2) ∧ (B = "true" ∨ B = "false") ∧

  ((Y[i] = 0 ∨ Y[j] = 0) ⇒ B = "true") ∧ next ≠ 1 ∧ next = 1] = "false".

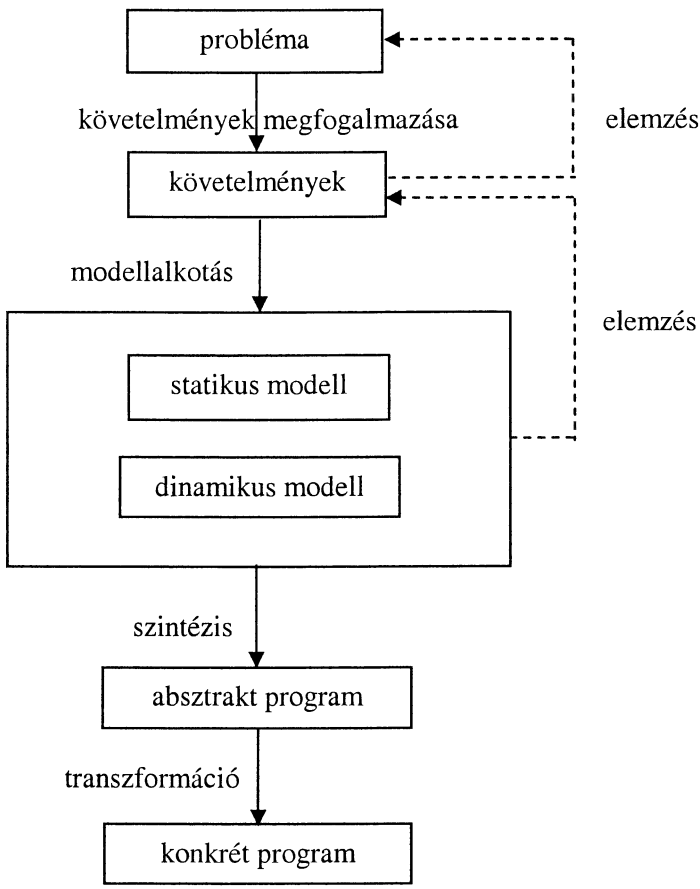
- Nem lehetséges, hogy S_2 vár és S_1 futását befejezte:

  [i ≥ 0 ∧ j ≥0 ∧ (next = 1 ∨ next = 2) ∧ (B = "true" ∨ B = "false") ∧

  ((Y[i] = 0 ∨ Y[j] = 0) ⇒ B = "true") ∧ next ≠ 2 ∧ next = 2] = "false".

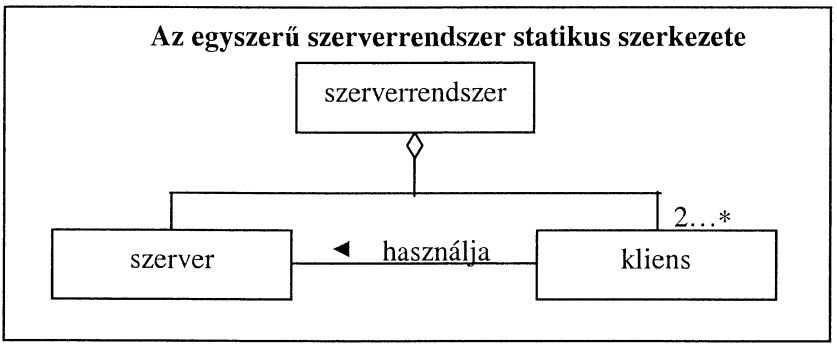
8. EGY MÓDSZER PÁRHUZAMOS PROGRAMOK ELŐÁLLÍTÁSÁRA

Adva van a párhuzamos rendszerrel szemben támasztott követelmények leírása. A feladat a párhuzamos rendszer előállítása. Ennek a végrehajtása a következő programfejlesztési modell alapján történhet.

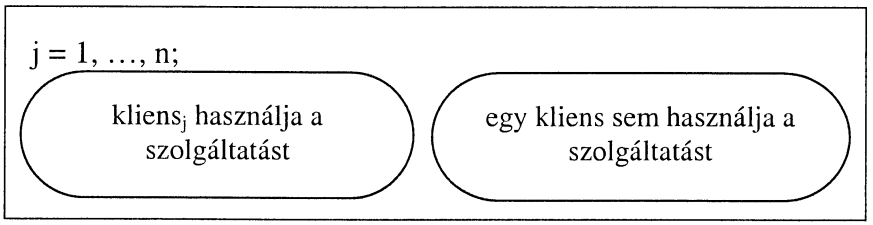


A modellalkotás lépéseit az egyszerű szerverrendszer példáján mutatjuk be. Legyen a rendszerrel szemben támasztott követelmény a következő: Adva van egy erőforrás, s az erre vonatkozó műveleteket egy szerver nyújtja a klienseknek. A szerver egyszerre csak egy klienst tud kiszolgálni.

1. A követelmények elemzése alapján meghatározzuk a rendszer közös erőforrását és annak tulajdonságait. Erőforrás a szerver szolgáltatása, amely kizárólagos használatú erőforrás.
2. A követelmények elemzése alapján meghatározzuk a rendszer közös erőforrásért versengő folyamatait: $kliens_1$, ..., $kliens_n$.
3. Megkonstruáljuk a rendszer statikus szerkezetét. A követelmények elemzése alapján azonosítjuk az objektumosztályokat, meghatározzuk az osztályok közötti relációkat a feladat megoldása során. Meghatározzuk a relációk tulajdonságait, paramétereit. Elkészítjük a megoldás modelljének szerkezetét, az osztálydiagramot:



4. A rendszer lehetséges állapotainak meghatározása:



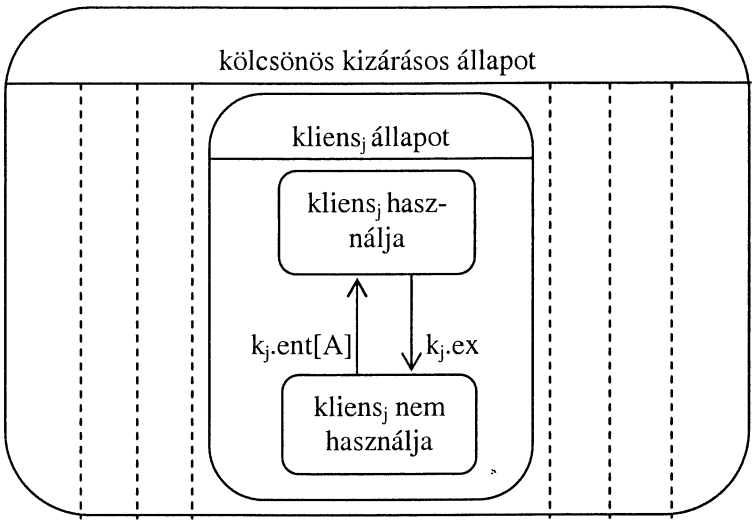
5. A rendszer állapotai között az állapotátmenetek meghatározása.

$$\text{kliens}_j\text{.ent; kliens}_j\text{.ex, } j = 1, \ldots, n.$$

6. A rendszer állapotai között az állapotátmenetek feltételeinek a meghatározása. Például: A kliens_j.ent állapotátmenet feltétele:

$$A: (\sim\exists k,\ 1 \le k \le n)(\textbf{in}\ \text{kliens}_k\ \text{használja}).$$

7. A rendszer állapotdiagramjának megkonstruálása.



AZ ABSZTRAKT PROGRAM MEGHATÁROZÁSÁNAK LÉPÉSEI

A rendszer állapotinvariánsának a meghatározása

1. Változók bevezetése a rendszer állapotainak a jellemzésére. I_j, $j = 1, 2, \ldots, n$. $I_j = 0$, ha a j-edik kliens nem használja, és $I_j = 1$, ha használja a szervert.
2. Az állapotok invariánsainak meghatározása a változókkal. A j-edik kliens esetén:

$$(\forall i,\ 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge I_j = 1;$$

illetve

$$(\forall i,\ 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge I_j = 0;$$

3. A rendszer szinkronizációs követelményeinek megfelelő rendszerállapot felírása a bevezetett változókkal. Csak egy kliens használhatja egyidejűleg az erőforrást:

$$(\forall i,\ 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge (0 \le I_1 + \ldots + I_n \le 1);$$

Az állapotjellemzők kezdőértékeinek a meghatározása

$$I_i = 0,\ i = 1, 2, \ldots, n.$$

A párhuzamos program vázlatának elkészítése

$$I_1 \leftarrow 0; \ldots; I_n \leftarrow 0;\ \textbf{parbegin}\ proc_1 \parallel \ldots \parallel proc_n\ \textbf{parend};$$

1. A folyamatok vázlatának elkészítése.

```
procj: while "true" do
        nem kritikus szakaszj;
        kritikus szakaszj;
                        od;
j = 1, ..., n;
```

2. Az oszthatatlan utasítások meghatározása a kritikus szakaszok számára.

$$\langle I_j \leftarrow 1 \rangle,\ j = 1, \ldots, n;$$

$$\langle I_j \leftarrow 0 \rangle,\ j = 1, \ldots, n;$$

3. A folyamatok vázlatának felírása oszthatatlan utasításokkal.

```
procj: while "true" do nem kritikus szakaszj;
<Ij ← 1>;      kritikus szakaszj;      <Ij ← 0>;
                  od;   (j = 1, ..., n)
```

4. A párhuzamos rendszerben az erőforrásokhoz való hozzáférésnél az invariánsok megmaradási feltételeinek a meghatározása.

$$\Sigma I_j = 0: (I_1 + \ldots + I_n = 0).$$

5. Az invariánsok elhelyezése a párhuzamos programban.

$I_1 \leftarrow 0; \ldots; I_n \leftarrow 0;$

$\{(\forall i, 1 \leq i \leq n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \leq 1)\}$

parbegin

...

|| **while** "true" **do**

nem kritikus szakasz$_j$;

$\{(\forall i, 1 \leq i \leq n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \leq 1) \wedge I_j = 0\}$

$<I_j \leftarrow 1>;$

$\{(\forall i, 1 \leq i \leq n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \leq 1) \wedge I_j = 1\}$

kritikus szakasz$_j$;

$<I_j \leftarrow 0>;$

$\{(\forall i, 1 \leq i \leq n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \leq 1) \wedge I_j = 0\}$

od;

||

...

parend;

6. Az invariáns megmaradásának biztosításához szükséges feltételek elhelyezése a kritikus utasítások előtt.

$I_1 \leftarrow 0; \ldots; I_n \leftarrow 0;$

$\{(\forall i, 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \le 1)\}$

parbegin ... || **while** "true" **do** nem kritikus szakasz$_j$;

$\{(\forall i, 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \le 1) \wedge I_j = 0 \wedge \Sigma I_j = 0\}$

$<I_j \leftarrow 1>;$

$\{(\forall i, 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \le 1) \wedge I_j = 1\}$

kritikus szakasz$_j$;

$<I_j \leftarrow 0>;$

$\{(\forall i, 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \le 1) \wedge I_j = 0\}$

od;

|| ... **parend**;

7. Az oszthatatlan utasítások megvalósítása megvárakoztató utasításokkal.

while "true" **do**

nem kritikus szakasz$_j$;

$\{(\forall i, 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \le 1) \wedge I_j = 0\}$

await $\Sigma I_j = 0$ **then** $I_j \leftarrow 1$ **ta**;

$\{(\forall i, 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \le 1) \wedge I_j = 1\}$

kritikus szakasz$_j$;

await "true" **then** $I_j \leftarrow 0$ **ta**;

$\{(\forall i, 1 \le i \le n)(I_i = 0 \vee I_i = 1) \wedge (I_1 + \ldots + I_n \le 1) \wedge I_j = 0\}$

od;

Kész az absztrakt program, amely az invariánsok nélkül a következő:

$I_1 \leftarrow 0; \ldots; I_n \leftarrow 0;$

parbegin ... || **while** "true" **do**

nem kritikus szakasz$_j$;

await $\Sigma I_j = 0$ **then** $I_j \leftarrow 1$ **ta**;

kritikus szakasz$_j$;

await "true" **then** $I_j \leftarrow 0$ **ta**;

od; || ... **parend**;

A KONKRÉT PROGRAM ELŐÁLLÍTÁSA

1. Szemaforok bevezetése. Ez most a mi esetünkben egyszerű feladat:

$$s = 1 - \Sigma I_j.$$

2. A szemaforok kezdőértékeinek a meghatározása:

$$s \leftarrow 1;$$

3. A megvárakoztató utasítások megvalósítása szemaforokkal:

await $s > 0$ **then** $s \leftarrow s - 1$; $I_j \leftarrow 1$; **ta**;

kritikus szakasz;

await "true" **then** $s \leftarrow s - 1$; $I_j \leftarrow 0$ **ta**;

4. A segédváltozók törlése, és a megvárakoztató utasítások P(s), V(s) utasításokkal való implementálása.

$s \leftarrow 1;$

parbegin ... || **while** "true" **do** nem kritikus szakasz;

P(s);

kritikus szakasz;

V(s);

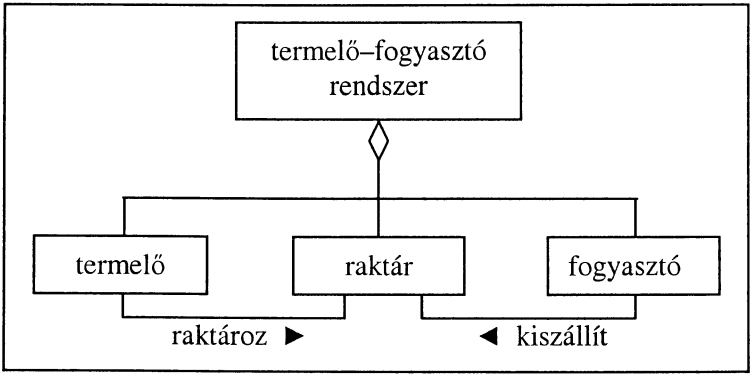
od; || ... **parend**;

8.1. A TERMELŐ–FOGYASZTÓ RENDSZER

Párhuzamos program készítendő a következő feladat megoldására. A termelő árut állít elő. Amikor az elkészült, elhelyezi a raktárban. Ezután kezdi elölről a termelést. A fogyasztó kiszállítja az árut a raktárból, majd feldolgozza azt. Amikor a feldolgozással végzett, kezdi a munkáját elölről.

Az egyszerűség kedvéért a raktárban egy egységnyi árunak legyen helye.

1. A követelmények elemzése alapján meghatározzuk a rendszer közös erőforrását és annak tulajdonságait. A közös erőforrás a raktár, amelyben egy hely van. Ez egy oszthatatlan erőforrás és így egy változóval valósítható meg. Legyen ennek a neve: R.
2. A követelmények elemzése alapján meghatározzuk a rendszer közös erőforrásért versengő folyamatait: Az erőforrást egy termelő folyamat és egy fogyasztó folyamat kívánja használni. Legyen ezek neve rendre termelő, fogyasztó.
3. Készítsük el a rendszer osztálydiagramját.

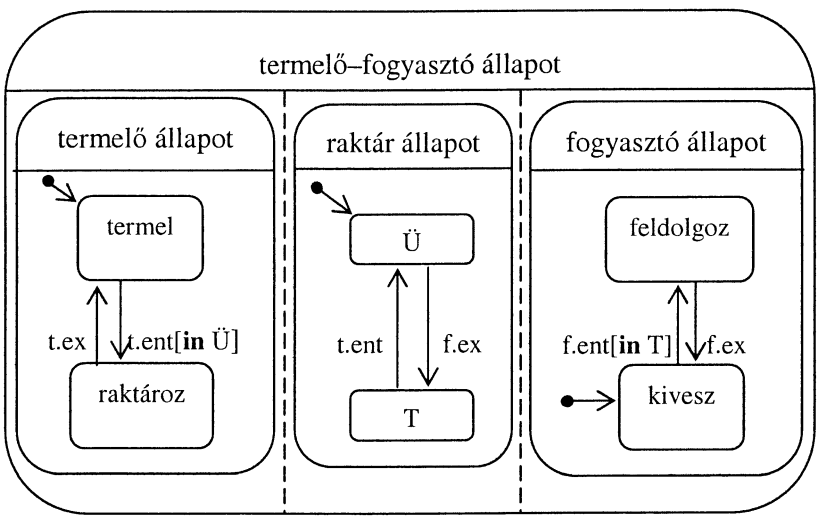


4. A rendszer lehetséges állapotainak meghatározása. A raktár állapota most üres állapot vagy tele állapot lehet. A termelő és a fogyasztó folyamatok ál-

lapotait az erőforráshoz való viszonyuk határozza meg. Ennek megfelelően az állapotok:

- a termelő állapotai: termel; raktároz;
- a fogyasztó állapotai: feldolgoz; kivesz;
- a raktár állapotai: tele; üres; jelölje ezeket rendre: T; Ü.

5. A rendszer állapotai között az állapotátmenetek meghatározása. A termelő ill. a fogyasztó raktárba való belépései ill. raktárból történő kilépései eredményezik az állapotváltozásokat. Legyenek ezek: t.ent; t.ex; f.ent; f.ex;
6. A rendszer állapotai között az állapotátmenetek feltételeinek a meghatározása. Mindkét folyamat akkor léphet be, ha a raktárt nem használja senki, és a termelő akkor helyezhet el árut a raktárban, ha a raktár üres: t.ent[**in** Ü]; a fogyasztó akkor szállíthat ki árut, ha a raktár tele van: f.ent[**in** T];
7. A rendszer állapotdiagramjának felépítése.



Elkészítettük tehát a probléma megoldásának modelljét UML-ben. A feladat most először az, hogy ennek a modellnek az alapján készítsük el a probléma egy absztrakt programját.

AZ ABSZTRAKT PROGRAM MEGHATÁROZÁSÁNAK A LÉPÉSEI

A rendszer állapotinvariánsának a meghatározása.

1. Változók bevezetése a rendszer állapotainak a jellemzésére. A termelő állapotát jellemezze tá, amelynek értéke 0, ha a termelő termel, és 1, ha raktároz. A fogyasztó állapotát jellemezze fá, amelynek értéke 0, ha a fogyasztó feldolgoz, és 1, ha kiszállít a raktárból. A raktár állapotát jellemezze rá, amelynek értéke 0, ha üres a raktár, és 1, ha tele van.
2. Az állapotok invariánsainak meghatározása a változókkal.

$$(\text{fá} = 0 \vee \text{fá} = 1) \wedge (\text{tá} = 0 \vee \text{tá} = 1) \wedge (\text{rá} = 0 \vee \text{rá} = 1);$$

3. A szinkronizáció követelményeinek megfelelő rendszerállapot felírása a bevezetett változókkal.

$$\text{TF: } (\text{tá} = 1 \wedge \text{rá} = 0 \wedge \text{fá} = 0) \vee (\text{tá} = 0 \wedge \text{rá} = 1 \wedge \text{fá} = 1) \vee$$

$$((\text{tá} = 0 \wedge \text{fá} = 0) \wedge (\text{rá} = 0 \vee \text{rá} = 1));$$

Az állapotjellemzők kezdőértékeinek a meghatározása

$$\text{fá} = 0; \text{tá} = 0;$$

A párhuzamos program vázlatának elkészítése

fá ← 0 ; tá ← 0;
{TF}
parbegin termelő || fogyasztó **parend**,

1. A folyamatok vázlatának elkészítése. Mindkettő egyszerű végtelen ciklus:

termelő:	fogyasztó:
while "true" **do**	**while** "true" **do**
x termék előállítása;	y ← R;
R ← x; **od**;	y termék feldolgozása; **od**;

2. Az oszthatatlan utasítások meghatározása a kritikus szakaszok számára.

$$\langle \text{tá} \leftarrow 1\rangle;\ \langle \text{tá} \leftarrow 0;\ \text{rá} \leftarrow 1\rangle;\ \langle \text{fá} \leftarrow 1\rangle;\ \langle \text{fá} \leftarrow 0;\ \text{rá} \leftarrow 0\rangle;$$

3. A folyamatok vázlatának felírása oszthatatlan utasításokkal.

termelő:	fogyasztó:
while "true" **do**	**while** "true" **do**
x termék előállítása;	$\langle \text{fá} \leftarrow 1\rangle$;
$\langle \text{tá} \leftarrow 1\rangle$;	$y \leftarrow R$;
$R \leftarrow x$;	$\langle \text{fá} \leftarrow 0;\ \text{rá} \leftarrow 0\rangle$;
$\langle \text{tá} \leftarrow 0;\ \text{rá} \leftarrow 1\rangle$; **od**;	y termék feldolgozása; **od**;

4. A párhuzamos rendszerben az erőforrásokhoz való hozzáférésnél az invariánsok megmaradási feltételeinek a meghatározása. A termelő beléphet, ha a raktár üres és a fogyasztó nem használja azt:

$$\text{rá} = 0 \wedge \text{fá} = 0.$$

A fogyasztó beléphet, ha a raktár tele van és termelő nem használja azt:

$$\text{rá} = 1 \wedge \text{tá} = 0.$$

5. Az invariánsok elhelyezése a párhuzamos programban.

termelő: $\{TF\}$	fogyasztó: $\{TF\}$
while "true" **do** x termék előállítása;	**while** "true" **do** $\{TF \wedge \text{fá} = 0\}$
$\{TF \wedge \text{tá} = 0\}$	$\langle \text{fá} \leftarrow 1\rangle$;
$\langle \text{tá} \leftarrow 1\rangle$;	$\{TF \wedge \text{fá} = 1\}$
$\{TF \wedge \text{tá} = 1\}$	$y \leftarrow R$;
$R \leftarrow x$;	$\langle \text{fá} \leftarrow 0;\ \text{rá} \leftarrow 0\rangle$;
$\langle \text{tá} \leftarrow 0;\ \text{rá} \leftarrow 1\rangle$;	$\{TF \wedge \text{fá} = 0\}$
$\{TF \wedge \text{tá} = 1\}$ **od**;	y termék feldolgozása; **od**;

6. Az invariáns megmaradásának biztosításához szükséges feltételek elhelyezése a kritikus utasítások előtt.

   A termelő esetében: $\{TF \wedge tá = 0 \wedge (fá = 0 \wedge rá = 0)\}\ \langle tá \leftarrow 1\rangle$;

   A fogyasztó esetében: $\{TF \wedge fá = 0 \wedge (tá = 0 \wedge rá = 1)\}\ \langle fá \leftarrow 1\rangle$;

7. Az oszthatatlan utasítások megvalósítása megvárakoztató utasításokkal.

   A termelő esetében: **await** (fá = 0 ∧ rá = 0) **then** tá ← 1**ta**;

   await "true" **then** tá ← 0; rá ← 1**ta**;

   A fogyasztó esetében: **await** (tá = 0 ∧ rá = 1) **then** fá ← 1 **ta**;

   await "true" **then** fá ← 0; rá ← 0 **ta**;

termelő:	fogyasztó:
{TF}	{TF}
while "true" **do**	**while** "true" **do**
x termék előállítása;	{TF ∧ fá = 0}
{TF ∧ tá = 0}	**await** (tá = 0 ∧ rá = 1) **then**
await (fá = 0 ∧ rá = 0) **then**	fá ← 1 **ta**;
tá ← 1 **ta**;	{TF ∧ fá = 1}
{TF ∧ tá = 1}	y ← R;
R ← x;	**await** "true" **then**
await "true" **then**	fá ← 0; rá ← 0 **ta**;
tá ← 0; rá ← 1**ta**;	{TF ∧ fá = 0}
{TF ∧ tá = 1}	y termék feldolgozása;
od;	**od**;

Kész az absztrakt program, amely az invariánsok nélkül a következő:

termelő:	fogyasztó:
while "true" **do**	**while** "true" **do**
x termék előállítása;	**await** (tá = 0 ∧ rá = 1) **then**
await (fá = 0 ∧ rá = 0) **then**	fá ← 1 **ta**;
tá ← 1 **ta**;	y ← R;
R ← x;	**await** "true" **then**
await "true" **then**	fá ← 0; rá ← 0 **ta**;
tá ← 0; rá ← 1 **ta**;	y termék feldolgozása;
od;	**od**

A KONKRÉT PROGRAM ELŐÁLLÍTÁSA

1. Szemaforok bevezetése. Két szemaforra van szükségünk. Kézenfekvő választás

   a termelő részére: t = 1 - (fá + rá);

   a fogyasztó részére: f = tá + rá;

2. A szemaforok kezdőértékeinek a meghatározása

   t ←1; f ← 0;

3. A megvárakoztató utasítások megvalósítása szemaforokkal.

   await t > 0 **then** t ← t - 1 **ta**;

   await "true" **then** t ← t + 1 **ta**;

   await f > 0 **then** f ← f - 1 **ta**;

   await "true" **then** f ← f + 1 **ta**;

4. A segédváltozók törlése, és a megvárakoztató utasítások implementálása P(s), V(s), utasításokkal.

Kész a konkrét program, amely az invariánsok nélkül a következő:

termelő:	fogyasztó:
while "true" **do**	**while** "true" **do**
x termék előállítása;	P(f);
P(t);	y ← R;
R ← x;	V(t);
V(f);	y termék feldolgozása;
od;	**od**;

8.2. ABSZTRAKT MEGVÁRAKOZTATÓ UTASÍTÁSOK MEGVALÓSÍTÁSA KONKRÉT SZEMAFORUTASÍTÁSOKKAL

Az eddigi példák esetében azt láttuk, hogy a konkrét program megvalósítása során a szemaforok meghatározása viszonylag egyszerű feladat volt. A segédváltozókból kézenfekvő ötlettel, egyszerű formula alapján elő lehetett állítani a szemaforokat. Összetettebb feladatok esetén már ennek megoldása némiképp nehezebb feladatot jelent. Célszerű tehát általános módszert adni a probléma megoldására.

Legyen adva az S absztrakt párhuzamos program, amelyben atomi utasítások vannak:

await β_j **then** S_j **ta**; j = 1, ... ,n; szinkronizációs utasítások, ill.

await "true" **then** S_j **ta**; j = 1, ..., n; megszakíthatatlan utasítások.

A feladat az, hogy ezeket az utasításokat valósítsuk meg Dijkstra-féle szemaforutasításokkal.

Ezek az utasítások atomi utasítások. A megvalósítás során tehát egy ilyen utasítás megvalósításának is atomi utasításnak kell lennie. Egy ilyen utasítás atomi végrehajtásához tehát vezessünk be egy s szemafort, amelynek kezdőértéke s = 1. Így az egyszerű **await** β_j **then** S_j **ta**; megvalósítása Dijktsra kölcsönös kizárási modellje alapján a következő lehet:

```
P(s);
Sj
V(s);
```

A probléma abból adódik, hogy egy ilyen utasítás végrehajtása után gondoskodni kell a várakozó folyamatok indításáról is. Kell tehát egy beütemezési algoritmus is az atomi utasítás befejezéseként. A kellően általános megoldás tehát a következő lesz:

```
P(s);
Sj
call beütemezés;
```

Ennél a megoldásnál a V(s) utasítás kiadására a beütemezési algoritmus keretében kerülhet sor. Erre a kérdésre később térünk vissza. Vizsgáljuk előbb a megvárakoztató utasítás általánosabb formáját. Szemantikailag azonos β_j-nél több folyamat is várakozhat. Ezeket a folyamatokat be kell állítani a β_j-hez tartozó sorba, majd a beütemezési stratégia alapján dönteni kell arról, hogy a várakozó sorok közül melyikből kapja meg a vezérlést egy folyamat. Ehhez a megvárakoztatáshoz rendeljünk minden egyes β_j-hez egy bj szemafort. A szemafor kezdőértéke: $b_j = 0$, mivel minden folyamatnak be kell előbb állnia egy sorba, mielőtt a beütemezési stratégia alapján engedélyt kap a további futásra. Célszerű

egy külön változóban nyilvántartani a sorban várakozók számát. Legyen ez a változó c_j, amelynek kezdőértéke: $c_j = 0$.

Ezek után az **await** β_j **then** S_j **ta**; utasítás megvalósítása a következő lesz:

```
P(s);
if ~βj then cj ← cj + 1; V(s); P(bj) fi;
Sj;
call beütemezés;
```

A következő feladat tehát a beütemezési algoritmus elkészítése. Gyakran előfordul, hogy adott a prioritás. Tegyük fel tehát, hogy a β_j–nél várakozó sorok prioritása j-vel azonos. A szemantikailag azonos β_j-nél várakozókat az elindítás szempontjából egyenrangúaknak tekintjük. Ekkor a beütemezési stratégia a következő lehet:

```
procedure beütemezés(n: nat) =
        j ← n; B ← "false"
            while 1 ≤ j ∧~B do
    if βj ∧ cj > 0 then cj ← cj – 1; V(bj); B ← "true" fi;
                                        od;
```

Ha nincs megadva az egyes folyamattípusok prioritása, akkor nem determinisztikus beütemezési stratégiát alkalmazhatunk. A beütemezés algoritmusa ekkor a következő lesz:

procedure beütemezés(n: **nat**) =
if $\beta_1 \wedge c_1 > 0 \rightarrow c_1 \leftarrow c_1 - 1$; $V(b_1)$
□
...
□
$\beta_n \wedge c_n > 0 \rightarrow c_n \leftarrow c_n - 1$; $V(b_n)$
else V(s)
fi;

Nézzük ennek alkalmazását a következő példa esetében.

8.2.1. PÉLDA Adott az étkező filozófusok egy problémájának absztrakt megoldása:

eszik(1) $\leftarrow$ 0; ... ; eszik(m) $\leftarrow$ 0; **parbegin** fil_1 || ... || fil_m **parend;**
ahol fil_j: **while** "true" **do**
await eszik(j $\oplus$ 1) = 0 $\wedge$ eszik(j $\ominus$ 1) = 0 **then** eszik(j) $\leftarrow$ 1 **ta;**
étkezik;
await "true" **then** eszik(j) $\leftarrow$ 0 **ta**;
gondolkodik;
od;
itt $\oplus$, $\ominus$ moduló m összeadást ill. kivonást jelent; j = 1, ..., m;

Valósítsuk meg ezt az absztrakt megoldást szemaforutasításokkal úgy, hogy a beütemezés során a jobb kéz felül ülő filozófusnak mindig legyen elsőbbsége a tőle balra ülővel szemben, ha mindketten enni akarnak, de az m-edik helyen ülő filozófusnak legyen elsőbbsége mindkét szomszédjával szemben.

MEGOLDÁS A meg nem szakítható utasítások szemaforja: s.

$$\beta_j: \text{eszik}(j \oplus 1) = 0 \wedge \text{eszik}(j \ominus 1) = 0$$

a β_j feltételre várakozók szemaforja: b_j; a várakozók száma: c_j.

- A szemaforok kezdőértékeinek a meghatározása:

$$s \leftarrow 1;\ b_1 \leftarrow 0;\ \ldots\ ;\ b_m \leftarrow 0;\ c_1 \leftarrow 0;\ \ldots;\ c_m \leftarrow 0;$$

- A megvárakoztató utasítások megvalósítása szemaforokkal:

Az **await** eszik(j ⊕ 1) = 0 ∧ eszik(j ⊖ 1) = 0 **then** eszik(j) ← 1 **ta;** utasítás implementációja:

P(s); **if** eszik(j ⊕ 1) = 1 ∨ eszik(j ⊖ 1) = 1 **then** $c_j \leftarrow 1$; V(s); P(b_j) **fi**;
eszik(j) ← 1;
call beütemezés(m);

Az **await** "true" **then** eszik(j) ← 0 **ta**; utasítás implementációja:

P(s);
eszik(j) ← 0;
call beütemezés(m);

- A beütemezési eljárás:

procedure beütemezés(m) **begin** B←"true"; j ← m;
while j > 0 **do**
if (eszik(j ⊕ 1) = 0 ∧ eszik(j ⊖ 1) = 0) ∧ c_j = 1)
then $c_j \leftarrow 0$; V(b_j); B←"false" **fi**; j ← j - 1; **od**;
if B **then** V(s) **fi;** **end** beütemezés

8.2.2. PÉLDA Tekintsük ismét az adatbázis következő informális specifikációját.

- Az adatbázist n > 1 darab felújító és m > 1 darab lekérdező folyamat használja.
- A felújító folyamatok egymást kölcsönösen kizárják az adatbázis használatából.
- A lekérdező folyamatok közül tetszőleges számú folyamat használhatja egyidejűleg az adatbázist.
- A felújító folyamat és a lekérdező folyamat egymást kölcsönösen kizárják az adatbázis használatából.
- A felújítónak, ha az adatbázist használni kívánja, elsőbbsége van a lekérdező folyamatokkal szemben.

Végezzük el az adatbázis szintézisét!

MEGOLDÁS A követelmények elemzése alapján meghatározzuk a rendszer közös erőforrását és annak tulajdonságait. A közös erőforrás az adatbázis, amely felújító folyamat számára kizárólagos használatú erőforrás, a lekérdező folyamatok számára pedig megosztható erőforrás.

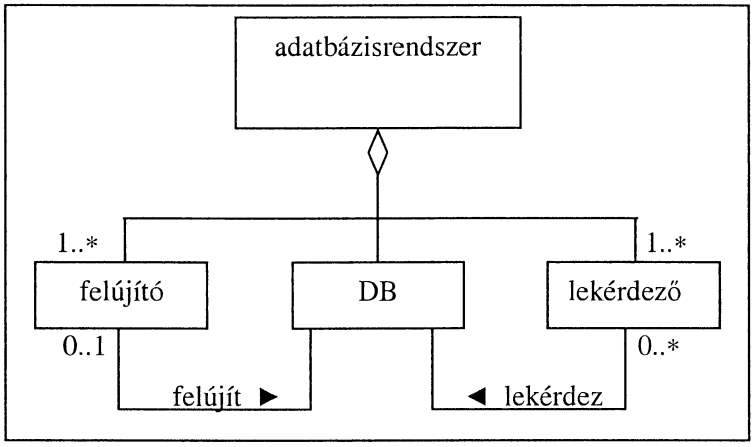
Legyen ennek a neve DB.

A követelmények elemzésével meghatározzuk a rendszer közös erőforrásért versengő folyamatait. Az erőforrást n darab felújító folyamat és m darab lekérdező folyamat használja. Legyen ezek azonosítója rendre:

$writer_i$, i = 1, ..., n;

$reader_j$; i = 1, ..., m;

1. Készítsük el a rendszer osztálydiagramját:



2. A rendszer lehetséges állapotainak meghatározása:

   Mindkét folyamat ciklikus.

   A felújító folyamat állapotai: nem ír, írni akar, ír.

   A lekérdező folyamat állapotai: olvas, nem olvas.

3. A rendszer állapotai között az állapotátmenetek meghatározása:

   A felújító ill. a lekérdező folyamat adatbázishoz való hozzáférése ill. az adatbázis elengedése eredményezi az állapotváltozásokat. Legyenek ezek:

   write.req; write.ent; write.ex; read.ent; read.ex;

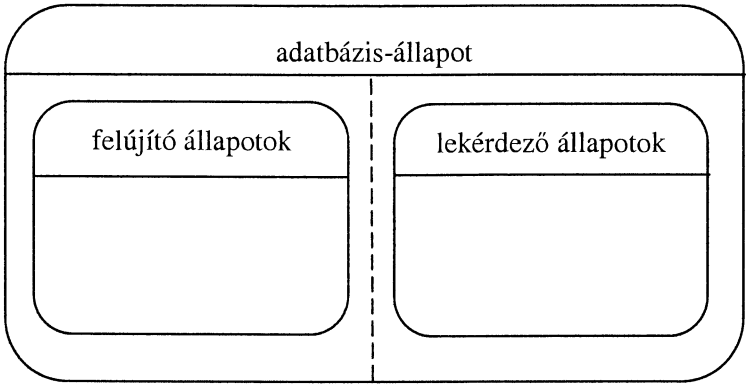
4. A rendszer állapotai közötti állapotátmenetek feltételeinek a meghatározása:

   A_j: $(\forall i)$(**in** nem $ír_i$)

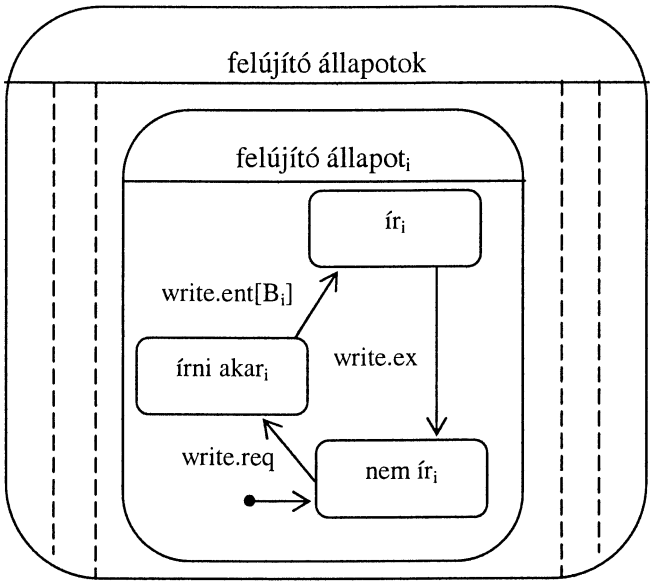
   B_i: $(\forall j)$(**in** nem $olvas_j$) $\wedge$ $(\sim\exists l, l \neq i)$(**in** $ír_l$);

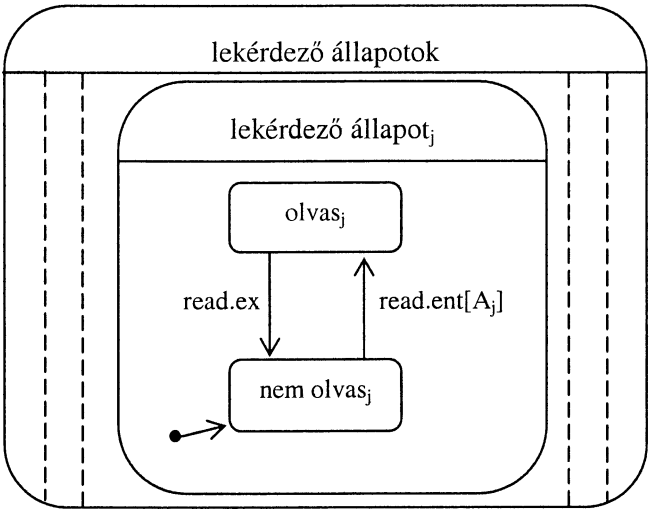
5. A rendszer állapotdiagramjának megkonstruálása**:**

   Az adatbázis-állapot egy állapotaggregáció, amelyet a felújító állapotok általánosítása és a lekérdező állapotok általánosítása alkot.



Egy felújító állapot$_i$ egy állapotáltalánosítás a következő konkrét állapotokkal:





AZ ABSZTRAKT PROGRAM MEGHATÁROZÁSÁNAK LÉPÉSEI

A rendszer állapotinvariánsának a meghatározása

1. Változók bevezetése a rendszer állapotainak a jellemzésére:
   - w = az adatbázist használó felújító folyamatok száma.
   - r = az adatbázist használó lekérdező folyamatok száma.
   - req = az adatbázis használatára igényt bejelentett felújítók száma.
2. Az állapotok invariánsainak meghatározása a változókkal:
   - felújító nem használja és lekérdező használja az adatbázist:

     $$(w = 0 \wedge r > 0);$$

   - felújító nem használja és lekérdező sem használja az adatbázist:

     $$(w = 0 \wedge r = 0);$$

   - felújító használja és lekérdező nem használja az adatbázist:

     $$(w = 1 \wedge r = 0);$$

3. A szinkronizáció szerint megfelelő állapotok invariánsa:

$$PFL: [(w = 0 \wedge r > 0) \vee (w = 0 \wedge r = 0) \wedge (w = 1 \wedge r = 0)] \wedge$$

$$[(w = 0 \vee w = 1) \wedge r \geq 0 \wedge req \geq 0]$$

4. Az állapotjellemzők kezdőértékeinek a meghatározása:

$$w = 0; r = 0; req = 0;$$

5. A párhuzamos rendszerben az erőforrásokhoz a szinkronizációs feltételnek megfelelő hozzáférés esetén az invariánsok megmaradási feltételeinek a meghatározása:

$$A_j: (\forall i)(\textbf{in} \text{ nem } ír_i): w = 0 \wedge req = 0;$$

$$B_i: (\forall j)(\textbf{in} \text{ nem } olvas_j) \wedge (\sim\exists l, l \neq i)(\textbf{in } ír_l): r = 0 \wedge w = 0;$$

Az absztrakt program:

$w \leftarrow 0; r \leftarrow 0; req \leftarrow 0;$

$\{PFL \wedge (w = 0 \wedge r = 0 \wedge req = 0)\}$

parbegin felújító$_1$ || ... || felújító$_n$ || lekérdező$_1$ || ... || lekérdező$_m$ **parend**;

lekérdező: **while** "true" **do**

{PFL}

await $w = 0 \wedge req = 0$ **then** $r \leftarrow r + 1$ **ta**;

$\{FL \wedge r > 0\}$

lekérdezés;

await "true" **then** $r \leftarrow r - 1$ **ta**;

{FL}

lekérdezés feldolgozása;

od;

felújító: **while** "true" **do**

{PFL}

felújítás előkészítése;

{PFL}

await "true" **then** req ← req + 1 **ta**;

await (w = 0 ∧ r = 0) **then** w ← w + 1; req ← req - 1 **ta**;

{PFL ∧ w > 0}

felújítás;

await "true" **then** w ← w – 1 **ta**;

{PFL}

od;

A KONKRÉT PROGRAM ELŐÁLLÍTÁSA

1. Szemaforok bevezetése:

   a meg nem szakítható utasítások szemaforja: s.

   βw: (w = 0 ∧ r = 0); szemafor: bw; számláló:cw.

   βr: (w = 0 ∧ req = 0); szemafor: br; számláló: cr.

2. A szemaforok kezdőértékének a meghatározása:

   s ← 1; bw ← 0; br ← 0; cw ← 0; cr ← 0; w ← 0; r ← 0;

3. A megvárakoztató utasítások megvalósítása szemaforokkal:

   Felújító esetén az

   await "true" **then** req ← req + 1 **ta**;

   és az

   await (w = 0 ∧ r = 0) **then** w ← w + 1; req ← req - 1 **ta**;

   implementációja a következő:

P(s);
if w > 0 ∨ r > 0 **then** cw ← cw + 1; V(s); P(bw) **fi**;
w ← w + 1;
call beütemezés;

await "true" **then** w ← w - 1 **ta**; implementációja:

P(s);
w ←w - 1;
call beütemezés;

Lekérdező esetén az **await** w = 0 ∧ req = 0 **then** r ← r + 1 **ta**; implementációja:

P(s);
if w > 0 ∨ cw > 0 **then** cr ← cr + 1; V(s); P(br) **fi**;
r ← r + 1;
call beütemezés;

await "true" **then** r ← r – 1 **ta**; implementációja:

P(s);
r ← r - 1;
call beütemezés;

A beütemezési eljárás:

```
procedure beütemezés =
    begin B ← "false";
            if (r = 0 ∧ w = 0) ∧ cw > 0
          then cw ← cw - 1; B ← "true"; V(bw) fi;
                        □
          if ((w = 0 ∧ cw = 0 ∧ cr > 0 ) ∧ ~B)
            then cr ← cr - 1; V(br); else V(s) fi;
    end beütemezés;
```

Végül a konkrét program a következő:

s ← 1; bw ← 0; br ← 0; cw ← 0; cr ← 0; w ← 0; r ← 0;

parbegin felújító$_1$ ||...|| felújító$_n$ || lekérdező$_1$ ||...|| lekérdező$_m$ **parend**;

```
while "true" do felújítás előkészítése;
                    P(s);
if w > 0 ∨ r > 0 then cw ← cw + 1; V(s); P(bw) fi;
                  w ← w + 1;
              call beütemezés;
                          felújítás;
                            P(s);
                          w ← w - 1;
              call beütemezés;
                od;
```

A lekérdező folyamat konkrét formája a következő:

```
while "true" do
                    P(s);
  if w > 0 ∨ cw > 0 then cr ← cr + 1; V(s); P(br) fi;
                    r ← r + 1;
                 call beütemezés;
                  lekérdezés;
                    P(s);
                    r ← r - 1;
                 call beütemezés;
              a lekérdezés feldolgozása;
            od;
```

9. OSZTOTT RENDSZEREK

Az **osztott rendszer** (distributed system) egymástól elkülönülten létező komponensek egy halmaza. Ezek a komponensek egymástól függetlenül dolgoznak felhasználva saját erőforrásukat, azonban időről időre üzeneteket küldenek egymásnak.

Az **osztott programrendszer** végrehajtása párhuzamosan futó folyamatoknak egy halmaza, amelyek egymással üzenetek révén kommunikálnak. Minden egyes folyamat rendelkezik saját változók egy halmazával, és a különböző folyamatokhoz tartozó változók halmazai diszjunktak.

Az üzenetformában történő kommunikáció lehet:

- **szinkron kommunikáció**. A küldő az üzenetet akkor tudja átadni, ha a fogadó az adott időpontban kész annak átvételére (telefonálás).
- **aszinkron kommunikáció**. A küldő az üzenetet akkor is el tudja küldeni, ha a fogadó éppen nem kész annak az átvételére (levélküldés).

SZINKRON KOMMUNIKÁCIÓ CSATORNÁN KERESZTÜL

Kommunikációs csatorna

- A kommunikációs csatorna nem irányított, azaz mindkét irányban használható.
- A kommunikációs csatornának nincsen típusa, azaz különböző típusú üzenetek továbbítására egyaránt alkalmas.

A kommunikációs csatorna I/O-utasításai

Input utasítás

- **Szintaxis:** C?x, ahol C a csatorna azonosítója egy adott *csat* halmazból, C∈ *csat*, x egyszerű változó.
- **Szemantika:** Annak a kívánságnak a kifejezése, hogy az utasítást kiadó programban az x változó egy értéket fogadjon a C csatornán keresztül. Ha ez bekövetkezik, akkor az x változó felveszi az értéket.

Output utasítás

- **Szintaxis:** C!e, ahol C a csatorna azonosítója, e annak a programnak lokális változóival képezett aritmetikai kifejezés, amely a C!e utasítást kiadja.
- **Szemantika:** Annak a kívánságnak a kifejezése, hogy az utasítást kiadó program az e kifejezés értékét elküldje a C csatornán keresztül.

Őrfeltétellel ellátott input utasítás

- **Szintaxis:** b;C?x; ill. b;C!e, ahol b egy logikai kifejezés.
- **Szemantika.** A I/O csatornautasítás kiegészül egy tesztelési függvénynyel, amely folyamatosan vizsgálja, hogy lehetséges-e a kommunikáció a C csatornán keresztül. Amíg ez nem következik be, a b kifejezés kiértékelése "false" értéket eredményez, ha bekövetkezik, akkor pedig "true" értéket.

  A "true";C?e ill "true";C!e rövid jelölése rendre C?e ;C!e.

A kommunikáció a b_1;C?x és b_2;C!e utasításpár esetén a következőt jelenti:

- Azt mondjuk, hogy a b_1;C?x input utasítás és a b_2;C!e output utasítás egymáshoz illeszthető, ha az e kifejezés típusa és az x változó típusa azonos, és mind b_1, mind b_2 kiértékelése igaz értéket szolgáltat.
- Mindkét kérés teljesítésének várnia kell, amíg a másik nem jelenik meg.

- Mindkét kérés végrehajtása egyszerre történik meg. (Szinkron üzenet).
- A kommunikáció eredménye az, hogy az x ← e értékadás megtörténik.

A programnyelv szintaxisa

⟨kifejezés⟩: e ::= ⟨konstans⟩ | ⟨változó⟩ | $e_1 + e_2$ | $e_1 - e_2$ | $e_1 * e_2$

⟨bool kifejezés⟩: b ::= "true"| ~b | $b_1 \vee b_2$ | $e_1 = e_2$ | $e_1 < e_2$

⟨utasítás⟩: s ::= $b \rightarrow x \leftarrow e$ | $\langle C!e \rightarrow x \leftarrow e_1 \rangle$ | $\langle b; C?x \rightarrow x \leftarrow e_1 \rangle$ | $S_1; S_2$ |

if $S_1 \square \ldots \square S_n$ **fi** | **do** $S_B \square S_E$ **od**

⟨értékadás⟩ : $x \leftarrow e ::= (x_1, \ldots, x_n) \leftarrow (e_1, \ldots, e_n)$

⟨program⟩: P ::= **parbegin** $S_1 \parallel \ldots \parallel S_n$ **parend**

Szintaktikai korlátozás:

S_i, S_j, S_E utasítások az **if** $S_1 \square \ldots \square S_n$ **fi** és a **do** $S_B \square S_E$ **od** összetett utasításokban mindig feltétellel ellátott utasítások, kivéve

- S lokális értékadás;
- S a következő csatornautasítás $\langle b;C?x \rightarrow x \leftarrow e_1 \rangle$;
- $S = S_1; S_2$; szekvencia és $S_1 = b_1 \rightarrow S_1'$;
- $S =$ **if** $S_1 \square \ldots \square S_n$ **fi** és $(\forall j, j \in \{1, \ldots, n\})(S_i = b_i \rightarrow S_{i'})$;

A csatornautasítások előfordulására vonatkozó korlátozások:

- $S_1; S_2$ esetén, ha S_1 tartalmazza a C!e csatornautasítást, akkor S_2 nem tartalmazhat C?x csatornautasítást.
- **if** $S_1 \square \ldots \square S_n$ **fi**; **do** $S_B \square S_E$ **od** utasítások esetén minden i, $j \in \{1, \ldots. n\}$, hasonlóan (S_B, S_E) párra, ha S_i tartalmazza a C!e csatornautasítást, akkor S_j nem tartalmazhat C?x csatornautasítást.
- **parbegin** $S_1 \parallel \ldots \parallel S_n$ **parend** esetén megköveteljük, hogy $var(S_j) \cap var(S_k) = \{\}$, ahol $\forall j, k \in \{1, \ldots, n\} \wedge j \neq k$.

Jelölés:

- change(S) azoknak az egyszerű változóknak a halmaza, amelyek az S folyamatban értékadás bal oldalán vagy inpututasításban szerepelnek.
- var(S) az összes változó, beleértve a csatornaneveket is.

DEFINÍCIÓ Két S_1és S_2 folyamatot diszjunktnak nevezünk, ha

$$change(S_1) \cap var(S_2) = var(S_1) \cap change(S_2) = \{\}.$$

DEFINÍCIÓ Azt mondjuk, hogy a C csatorna összeköti S_i-t és S_j-t, ha

$$C \in channel(S_i) \cap channel(S_j).$$

DEFINÍCIÓ A $S_1, \ldots, S_n$ folyamatokat pontról pontra összekötötteknek nevezzük, ha teljesül a következő feltétel:

$$(\forall i, j, k)(1 \leq i < j < k \leq n)(chanel(S_i) \cap channel(S_j) \cap chane(S_k) \neq \{\}).$$

DEFINÍCIÓ Osztott programrendszernek nevezzük az

$$S: \textbf{parbegin } S_1 \parallel \ldots \parallel S_n \textbf{ parend};$$

párhuzamos programot, ha

- az $S_1,\ldots,S_n$ folyamatok páronként diszjunktak és
- az $S_1,\ldots,S_n$ folyamatok pontról pontra összekötöttek.

Azt mondjuk, hogy S befejeződik, ha mindegyik S_j befejeződik.(Ha nincs végtelen ciklus, nincs hiba miatti leállás, nincs holtpont).

9.1. PÉLDA Tekintsük a termelő–fogyasztó rendszernek azt a változatát, amelyben a termelő egy X vektorból csatornán keresztül egyenként továbbítja az elemeket a fogyasztónak, amely azokat az érkezés sorrendjében egy Z vektorban helyezi el. Készítsük el a megoldás programját.

MEGOLDÁS

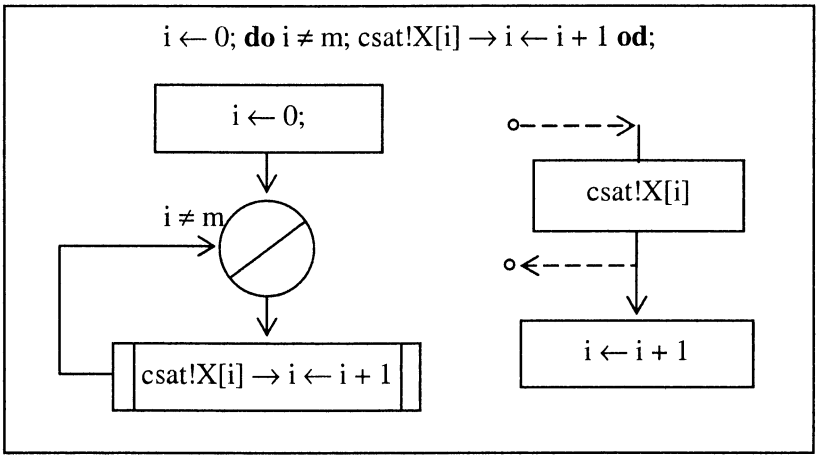
termelő–fogyasztó program: **parbegin** termelő || fogyasztó **parend**;

termelő: i ← 0; **do** i ≠ m; csat!X[i] → i ← i + 1 **od**;

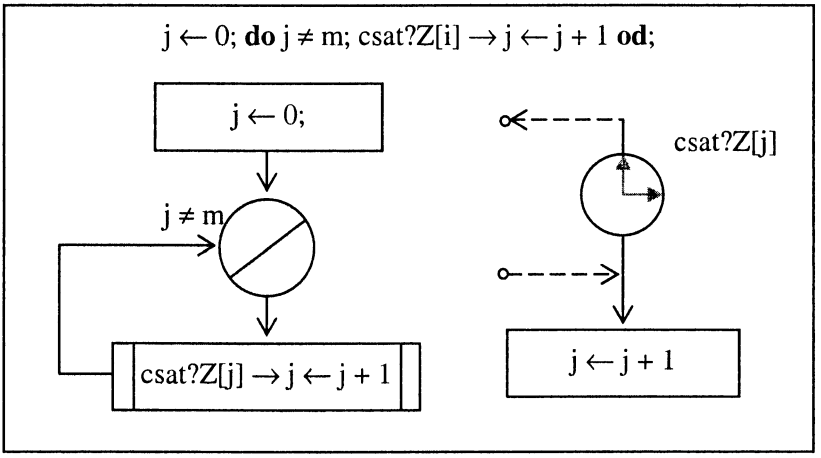
fogyasztó: j ← 0; **do** j ≠ m; csat?Z[j] → j ← j + 1 **od**;

Készítsük el a program végrehajtási gráfját UML-ben.

A termelő végrehatási gráfja:



A fogyasztó végrehajtási gráfja:



9.2. PÉLDA Iktassunk most a termelő és a fogyasztó közé egy továbbítót. Legyen ennek a továbbítónak az is a feladata, hogy a selejtes árut kiválogatja, és azt nem küldi el. Tegyük fel, hogy a selejtes áru az a kapott karakter, amely az $\perp$ üres karakterrel egyenlő. Amikor a termelő befejezte a küldést, akkor a csillag karakter átküldésével jelzi azt a továbbítónak.

MEGOLDÁS

SR: **parbegin** termelő || továbbító || fogyasztó **parend**;

termelő: **begin** $i \leftarrow 0$;

do $i \neq m$; $csat_A!X[i] \rightarrow i \leftarrow i + 1$ **od**; $x \leftarrow *$; $csat_A!x$ **end**;

továbbító: $be \leftarrow 0$; $ki \leftarrow 0$; $y \leftarrow \perp$;

do $y \neq *$;$csat_A?y \rightarrow$ **if** $y = \perp \rightarrow$ skip

$y \neq \perp \rightarrow Y[be] \leftarrow y$; $be \leftarrow be + 1$**fi**

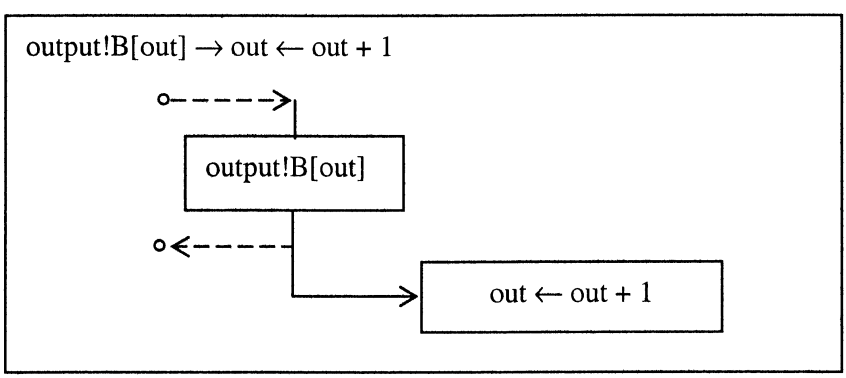
$ki \neq be$; $csat_B!Y[ki] \rightarrow ki \leftarrow ki + 1$

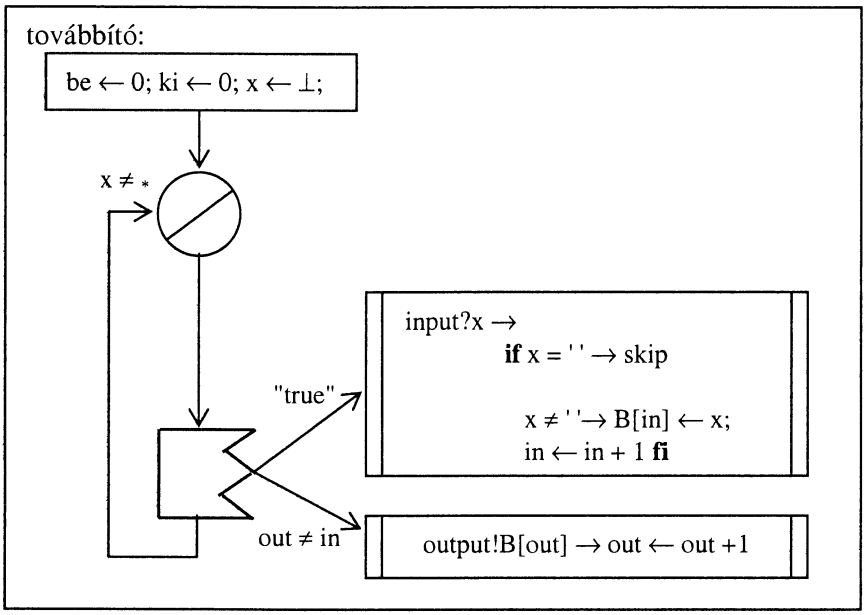
od;

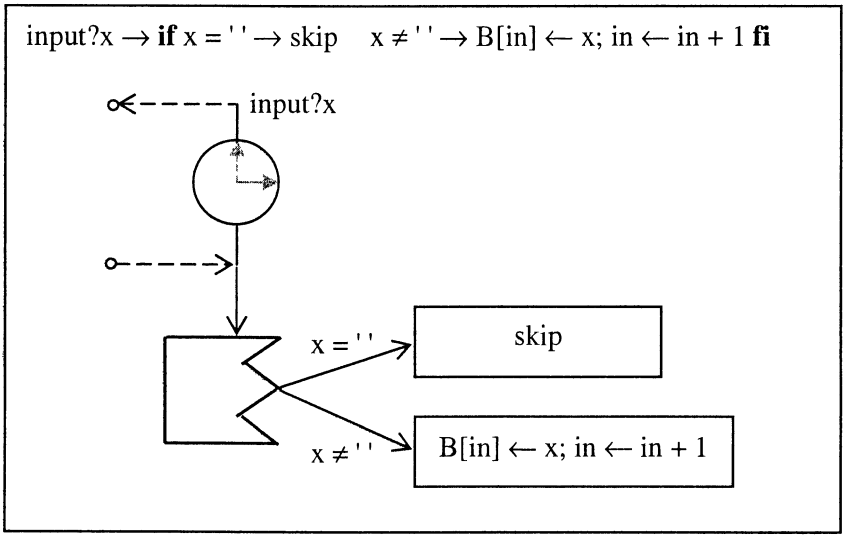
fogyasztó: $j \leftarrow 0$; $z = \perp$;

do $z \neq *$; $csat_B?z \rightarrow Z[j] \leftarrow z$; $j \leftarrow j + 1$ **od**;

A továbbító végrehajtási gráfját mutatják a következő ábrák:

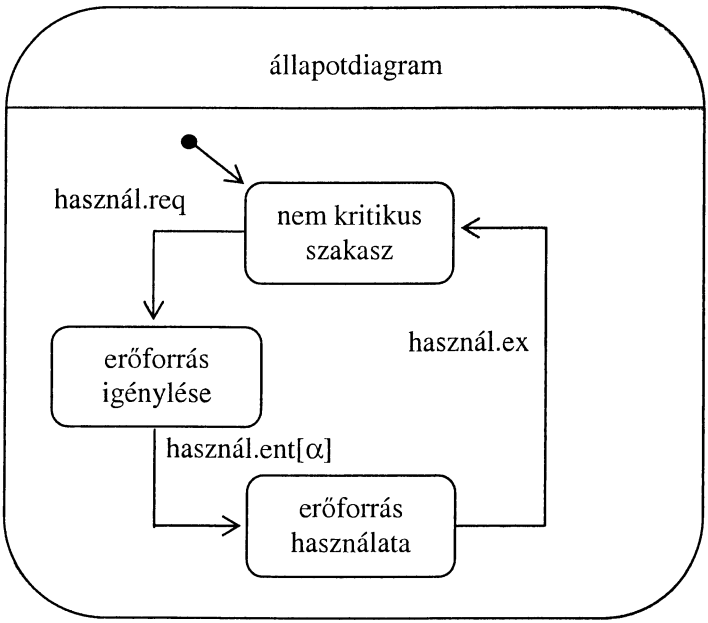



továbbító:
be ← 0; ki ← 0; x ← ⊥;
x ≠ *
input?x →
if x = ' ' → skip
x ≠ ' ' → B[in] ← x;
in ← in + 1 fi
"true"
out ≠ in
output!B[out] → out ← out +1


input?x → if x = ' ' → skip x ≠ ' ' → B[in] ← x; in ← in + 1 fi
input?x
x = ' '
skip
x ≠ ' '
B[in] ← x; in ← in + 1

10. OBJEKTUMOKHOZ VALÓ HOZZÁFÉRÉS SPECIFIKÁCIÓJA A SZINKRONIZÁCIÓ KÖVETELMÉNYEINEK MEGFELELŐEN

Párhuzamos programokban az objektum állapotának az objektum használata során bekövetkező változásait az alábbi állapotdiagrammal jellemezhetjük.



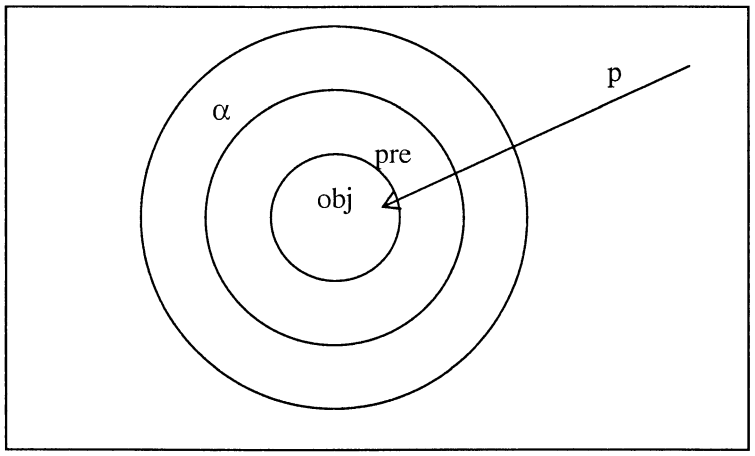
Az erőforrás használatának fázisai:

használ.req: igény bejelentése az erőforrásra.

használ[α].ent: az erőforrás lekötése, ahol α a szinkronizáció követelményeinek megfelelő belépési feltétel.

használ.ex: az erőforrás elengedése.

Az objektumot két védőgyűrű veszi körül. A külső védőgyűrű dinamikus, a belső pedig statikus belépési feltételt jelent az objektumon végzendő műveletek számára. A belső gyűrű, annak a feltételét tartalmazza, hogy a szóban forgó művelet az adott paraméterekkel elvégezhető-e az objektumon. A külső gyűrű pedig az adott pillanatban arra adja meg a választ, hogy a művelet a szikronizáció feltételei szerint hozzáférhet-e az objektumhoz. Ezt szemlélteti a következő ábra.



Az α-val jelölt feltétel időben változó feltétel. A pre szócskával jelölt feltétel csak akkor engedi végrehajtani a p műveletet, ha az értelmezve van az objektumra.

Az adattípus specifikációjának a következő formáját használtuk:

<típusnév> **is a type specification** =

parameters = ...;

exports =

class sorts:

oprs:

eqns:

...

Az **eqns** részben adjuk meg a statikus védőgyűrűre vonatkozó korlátozásokat. Nem konstrukciós függvény esetén például a következő axiómából adódik ez a korlátozás:

$$f_s(f_c(a)) = h(a) \Rightarrow pre(f_s(b)): b = f_c(a);$$

Konstrukciós művelet esetén pedig

$$attr(f_c(a)) > n \Rightarrow f_c(a) = \text{"undefined"} \Rightarrow pref_c(a): attr(a) \leq n.$$

alapján kapjuk meg.

Kézenfekvő tehát, hogy a dinamikus gyűrűre a szinkronizációnak megfelelő korlátozást is itt adjuk meg. Feladatunk tehát a szinkronizáció követelményeinek megfelelő axiómák megadása. Ezt a **syn** alapszó után fogjuk elhelyezni.

10.1. Egy absztrakt specifikációs nyelv a szinkronizáció követelményének leírására

Válasszuk erre a célra az elsőrendű predikátumkalkulus nyelvét. Definiáljunk ehhez a nyelvhez klózokat a következőképpen:

Megelőzési reláció Közösen használatos erőforrásra vonatkozó két műveleti fázis közötti reláció.

Műveleti fázis

- Igénybejelentés az erőforrásra; jele: <műveleti azonosító>.req.
- Az erőforrás igénybevétele; jele: <műveleti azonosító>.ent.
- Az erőforrás elengedése; jele: <műveleti azonosító>.ex.

MŰVELETI AZONOSÍTÓ A műveleti azonosító lehet a művelet neve, vagy a művelet neve és a művelet végrehajtásának sorszáma, és ezekhez társulhatnak még a műveletnek a szinkronizáció szempontjából fontos paraméterei.

A műveleti azonosító általános formája:

<művelet neve>[<sorszám>](<paraméter>).

A MEGELŐZÉSI RELÁCIÓ SZINTAXISA A → B.

A MEGELŐZÉSI RELÁCIÓ SZEMANTIKÁJA A → B = **if** az A műveleti fázis megelőzi a B műveleti fázist **then** "true" **else** "false".

AXIÓMÁK

- f[i].req → f[i].ent = "true".
- f[i].ent → f[i].ex = "true".
- f[i].req → f[i + 1].req = "true".

A MEGELŐZÉSI RELÁCIÓ ÁLTALÁNOSÍTÁSA

Szintaxis: R(x, y) ∧ (f[i](x).ex → g[j](y).ent), ahol R(x, y) a műveletekben a szinkronizáció szempontjából fontos szerepet játszó x, y paraméterek közötti reláció.

Szemantika: R(x, y) ∧ (f[i](x).ex → g[j](y).ent) = **ha** az f(x) művelet exit fázisa az i-edik végrehajtás esetén megelőzi a g(y) művelet enter fázisát a j-edik végrehajtás esetén, és a paramétereikre ugyanakkor fennáll az R(x, y) = "true" állítás is, **akkor** "true", **különben** "false".

Az univerzális és az egzisztenciális kvantor kétféle jelölése:

∀: minden (for all); ∃: valamely (for some).

Egy szinkronizáció specifikációjának kétféle felírási formája.

(∀j, k)(p[j].req → q[k].ent ⇒ p[j].ex → q[k].ent).

for all j, k: (p[j].req → q[k].ent ⇒ p[j].ex → q[k].ent).

MEGJEGYZÉS Célszerű lehet a következő rövidebb forma használata:

$$A \rightarrow B \wedge B \rightarrow C \equiv A \rightarrow B \rightarrow C;$$

A SZINKRONIZÁCIÓ KÖVETELMÉNYEINEK ALAPVETŐ SPECIFIKÁCIÓS ELEMEI A szinkronizáció specifikációja alapelemekből épül fel. A következőkben ezeket az elemeket vesszük sorba és megadjuk azok specifikációját az előzőekben ismertetett nyelvben.

1. **Kölcsönös kizárás.** A kölcsönös kizárás fogalmát a korábbiakban tárgyaltuk. A kölcsönös kizárás egy erőforrás és két folyamat viszonyára vonatkozott. Legyen ez a két folyamat p és q, és az erőforrást jelölje e.

   Az erőforrás állapotai: SZ: szabad; F: foglalt.

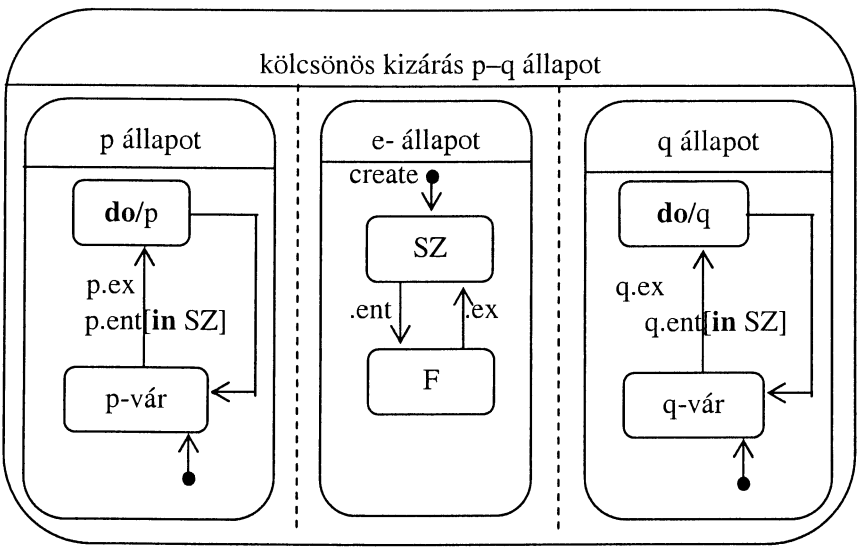
   A p folyamat állapotai:

   do/p: p használja az erőforrást; p-vár: p nem használja az erőforrást.

   A q folyamat állapotai:

   do/q: q használja az erőforrást; q-vár: q nem használja az erőforrást.

   Ekkor a kölcsönös kizárás állapotdiagramja a következő:



Készítsük el az állapotdiagram alapján a szinkronizáció specifikációját.

- Nem lehet ahhoz az adathoz hozzáférni, amit még nem hoztak létre.

$$(\forall i, j)(\text{create.ex} \rightarrow \text{q[j].ent} \wedge \text{create.ex} \rightarrow \text{p[i].ent}).$$

- Két művelet egymás kölcsönös kizárásának specifikációja:

$$(\forall i, j)(\text{p[i].ex} \rightarrow \text{q[j].ent} \vee \text{q[j].ex} \rightarrow \text{p[i].ent}).$$

- Az erőforráshoz való hozzáférés kölcsönös kizárásnak megfelelő rendszer specifikációja két folyamat esetén:

$$(\forall i, j)[(\text{create.ex} \rightarrow \text{q[j].ent} \wedge \text{create.ex} \rightarrow \text{p[i].ent}) \wedge$$
$$(\text{p[i].ex} \rightarrow \text{q[j].ent} \vee \text{q[j].ex} \rightarrow \text{p[i].ent})].$$

A kölcsönös kizárás speciális esete, amikor a két folyamat azonos, vagyis a p = q eset. Ekkor ugyanabból a műveletből nem végezhető több egyidejűleg az erőforráson. (Kölcsönös kizárás esete azonos műveletek esetén).

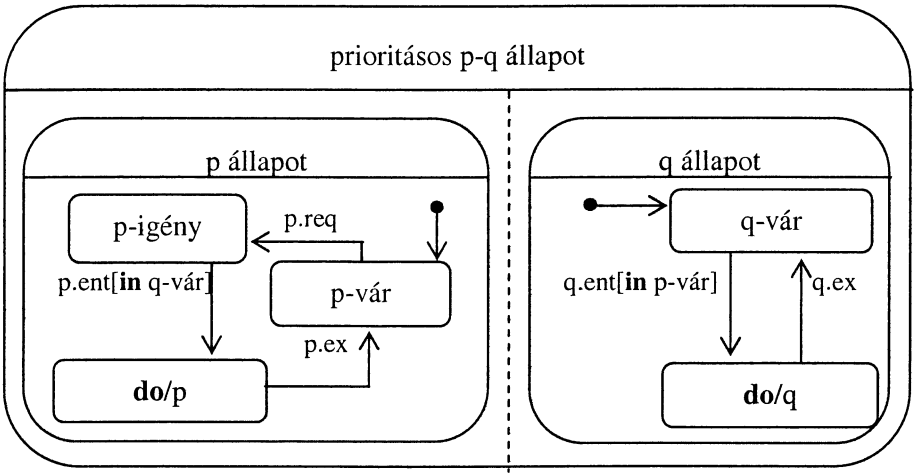
Specifikációja:

$$(\forall i)(\text{p[i].ex} \rightarrow \text{p[i + 1].ent}).$$

2. **A kölcsönös kizárás prioritásos esete**

- p és q egyszerre nem használhatják az erőforrást.
- a q művelet csak akkor végezhető az erőforráson, ha a p nem használja és nem is igényli azt.

Most az erőforrás állapotát nem tüntetjük fel újra az állapotdiagramban.



A p művelet prioritás miatti elsőbbsége:

$$(\forall i, j)(p[i].req \rightarrow q[j].ent \Rightarrow p[i].ex \rightarrow q[j].ent).$$

Kölcsönös kizárás:

$$(\forall i, j)(p[i].ex \rightarrow q[j].ent \vee q[j].ex \rightarrow p[i].ent).$$

A prioritásos kölcsönös kizárású rendszer specifikációja:

$$(\forall i, j)[(create.ex \rightarrow q[j].ent \wedge create.ex \rightarrow p[i].ent) \wedge$$

$$(p[i].ex \rightarrow q[j].ent \vee q[j].ex \rightarrow p[i].ent) \wedge$$

$$(p[i].req \rightarrow q[j].ent \Rightarrow p[i].ex \rightarrow q[j].ent]).$$

3. **Paraméterek közötti relációtól függő elsőbbség kölcsönös kizárás esetén.** Ha két folyamat akarja használni ugyanazt azt az erőforrást, de csak az egyik használhatja, akkor az férhet hozzá előbb, amelyiknek nagyobb a prioritása, feltéve, hogy a paramétereik között fennáll a R(x, y) reláció is. Legyen p a nagyobb, q pedig a kisebb prioritású folyamat.

$$R(x, y) \wedge (p(x).req \rightarrow q(y).ent) \Rightarrow p(x).ex \rightarrow q(y).ent).$$

4. **Beütemezés kölcsönös kizárás esetén.** Egy erőforrást használ egy p művelet. Közben érkezik két igénybejelentés az erőforrásra a q(x) és az r(y) mű-

veletek részéről. Ha R(x, y) igaz, akkor az erőforrást q(x) kapja meg előbb és r(y) pedig csak azután, ha q(x) elengedte azt.

$$(q(x).req \rightarrow p.ex \rightarrow q(x).ent) \wedge (r(y).req \rightarrow p.ex \rightarrow r(y).ent) \wedge R(x, y) \Rightarrow$$
$$q(x).ex \rightarrow r(y).ent$$

10.1.1. PÉLDA Készítsük el a kölcsönös kizárás adattípusának specifikációját párhuzamos környezet számára.

MEGOLDÁS

me(elem) **is a class specification** =

parameters =

sorts: elem

oprs: $\perp$: $\rightarrow$ elem

exports =

class sorts: me

oprs: create: $\rightarrow$ me

p: me elem $\rightarrow$ me

q: me $\rightarrow$ elem

eqns: m$\in$ me; e$\in$ elem;

q(create) = $\perp$

q(p(m, e)) = e

syn: for all i, j:

create $\rightarrow$ p[i].ent $\wedge$

p[i].ex $\rightarrow$ p[i + 1].ent $\wedge$

q[j].ex $\rightarrow$ q[j + 1].ent $\wedge$

(p[i].ex $\rightarrow$ q[j].ent $\vee$ q[j].ex $\rightarrow$ p[i].ent)

end me;

10.1.2. PÉLDA Készítsük el az adatbázis típusának specifikációját párhuzamos esetre.

MEGOLDÁS

db(elem) **is a class specification** =

parameters =

sorts: elem

oprs: $\perp$: $\rightarrow$ elem

exports =

class sorts: db

oprs: create: $\rightarrow$ db

write: db elem $\rightarrow$ db

read: db $\rightarrow$ elem

eqns: d$\in$me; e$\in$elem;

read(create) = $\perp$

read(write(d, e)) = e

syn: for all i, j:

create $\rightarrow$ writer[i].ent $\wedge$

write[i].ex $\rightarrow$ write[i + 1].ent $\wedge$

(writer[i].ex $\rightarrow$ reader[j].ent $\vee$

reader[j].ex $\rightarrow$ writer[i].ent) $\wedge$

(writer[i].req $\rightarrow$ read[j].ent $\Rightarrow$

writer[i].ex $\rightarrow$ reader[j].ent)

end db;

10.1.3. PÉLDA Készítsük el annak a veremnek a specifikációját, amelyet párhuzamos környezetben kívánunk használni.

MEGOLDÁS

stack(elem, n: **natbool**) **is a class specification** =

parameters = natbool +

sorts: elem

oprs: $\perp$: $\rightarrow$ elem

exports =

class sorts: stack

oprs:

create: $\rightarrow$ stack

push: stack elem $\rightarrow$ stack

pop: stack $\rightarrow$ stack

top: stack $\rightarrow$ elem

length: stack $\rightarrow$ natbool

eqns: s$\in$stack; e$\in$elem;

pop(create) = create

top(create) = $\perp$

length(create) = 0

pop(push(s, e)) = s

top(push(s, e)) = e

length(push(s, e)) = length(s) + 1

syn: for all i, j:

create.ex $\rightarrow$ push[i].ent $\wedge$

push[i].ex $\rightarrow$ pop[i].ent $\wedge$

pop[i].ex $\rightarrow$ push[i + n].ent $\wedge$

push[i].ex$\rightarrow$push[i + 1].ent $\wedge$

pop[i].ex→pop[i + 1].ent ∧

(pop[i].ex→push[j].ent ∨

push[j].ex → pop[i].ent) ∧

(pop[i].ex → top[j].ent ∨

top[j].ex → pop[i].ent) ∧

(top[i].ex → push[j].ent ∨

push[j].ex → top[i].ent) ∧

(length[i].ex → push[j].ent ∨

push[j].ex → length[i].ent) ∧

(length[i].ex → pop[j].ent ∨

pop[j].ex → length[i].ent)

end stack.

10.1.4. PÉLDA Szolgáltatás beütemezése a körgyűrű mentén. A körgyűrű mentén egy szervizkocsi teljesít szolgálatot. Szolgáltatásait csak a parkolókban lehet igénybe venni. Az egyszerűség kedvéért legyenek ezek a parkolók egymástól egységnyi távolságra elhelyezve. A szervizkocsi, ha valahol végzett a szolgáltatási munkájával, akkor a kérések közül mindig a hozzá legközelebb eső kérést teljesíti. Itt azonban feltesszük, hogy a szervizkocsi mindig egy irányban haladhat csak. Legyen az út mentén összesen n parkoló, amelyeket sorszámukkal azonosítunk.

MEGOLDÁS Határozzuk meg az A(z, x, y) feltételt, azaz annak a feltételét, hogy az z helyen javító gépkocsi, miután végzett, az x és y helyekről érkező kérések közül mikor választja az x helyet elsőnek.

Jelölje az x-edik helyen a j-edik javítást szerviz[j](x).

1. A feladat szerint egy szervizkocsi van, ezért javítás egyszerre csak egy lehet.

$(\forall j, k)(\text{szerviz}[j].ex \rightarrow \text{szerviz}[k].ent \vee \text{szerviz}[j].ex \rightarrow \text{szerviz}[i].ent)$.

2. A z helyen javít a szervizkocsi, és érkezik két kérés az x és az y helyről, amelyek közül egyiket előnyben részesíti a másikkal szemben, ha a z helyen végzett

$$\text{szerviz}[j](x).req \rightarrow \text{szerviz}[i](z).ex \rightarrow \text{szerviz}[j](x).ent \wedge$$
$$\text{szerviz}[k](y).req \rightarrow \text{szerviz}[i](z).ex \rightarrow \text{szerviz}[k](y).ent.$$

3. Az x helyről érkező kérés akkor részesül előnyben, ha az közelebb van a z helyhez. Ha egyforma távolságra vannak, akkor mindegy, hogy melyiket választja. A feltétel tehát:

$$|x - z| \leq |y - z|.$$

A megoldás a következő:

$$A(z, x, y): (\forall i, j, k, x, y, z)$$
$$((\text{szerviz}[i].ex \rightarrow \text{szerviz}[j].ent \vee \text{szerviz}[j].ex \rightarrow \text{szerviz}[i].ent) \wedge$$
$$(\text{szerviz}[j](x).req \rightarrow \text{szerviz}[i](z).ex \rightarrow \text{szerviz}[j](x).ent \wedge$$
$$\text{szerviz}[k](y).req \rightarrow \text{szerviz}[i](z).ex \rightarrow \text{szerviz}[k](y).ent) \wedge$$
$$(|x - z| \leq |y - z|) \Rightarrow$$
$$\text{szerviz}[j](x).ex \rightarrow \text{szerviz}[k](y).ent)$$

Prioritás módosítása: Tegyük fel, hogy a 0. helyről érkező kérésnek előnye van mindenkivel szemben. Tehát akkor annak a feltétele, hogy x legyen a kedvezményezett, legyen a következő:

- Sem x, sem y nem azonos a prioritással bíró hellyel, a két kérés a javítás helyétől azonos távolságból érkezik, vagy az x hely közelebb van, akkor az x helyről érkező kérés lesz előbb kiszolgálva. Ez a feltétel tehát formálisan a következő:

$$(x \neq 0 \wedge y \neq 0 \wedge |x - z| \leq |y - z|) \Rightarrow \text{szerviz}[j](x).ex \rightarrow \text{szerviz}[k](y).ent).$$

- Ha viszont x a prioritásos hely, akkor az lesz előbb kiszolgálva. Ez a kérés helyének feltétele alapján formálisan:

$$(x = 0 \Rightarrow \text{szerviz[j](x).ex} \rightarrow \text{szerviz[k](y).ent}).$$

Így a végeredmény:

$$A(z, x, y): (\forall i, j, k, x, y, z)$$
$$((\text{szerviz[i].ex} \rightarrow \text{szerviz[j].ent} \vee \text{szerviz[j].ex} \rightarrow \text{szerviz[i].ent}) \wedge$$
$$(\text{szerviz[j](x).req} \rightarrow \text{szerviz[i](z).ex} \rightarrow \text{szerviz[j](x).ent} \wedge$$
$$\text{szerviz[k](y).req} \rightarrow \text{szerviz[i](z).ex} \rightarrow \text{szerviz[k](y).ent}) \wedge$$
$$(((x \neq 0 \wedge y \neq 0 \wedge |x - z| \leq |y - z|) \vee x = 0) \Rightarrow$$
$$\text{szerviz[j](x).ex} \rightarrow \text{szerviz[k](y).ent}).$$

10.1.5. PÉLDA Készítsük el annak a halmaznak a specifikációját, amelyet párhuzamos környezetben kívánunk használni.

MEGOLDÁS

set (elem, n: **natbool**) **is a class specification** =

parameters = natbool +

sorts: elem

oprs: $\perp$: $\rightarrow$ elem

=: elem elem $\rightarrow$ bool

eqns: a, b, c$\in$elem;

$a = b \wedge b = c \Rightarrow a = c$

a = b = "true" $\Rightarrow$ b = a = "true"

exports =

class sorts: set

oprs: empty: $\rightarrow$ set

insert: set elem → set

delete: set elem → set

has: set elem → bool

size: set → natbool

eqns: s∈ set; e, e'∈ elem;

delete(empty, e) = empty

has(empty, e) = "false"

size(empty) = 0

delete(insert(s, e), e') = **if** e = e' **then** delete(s, e')

else insert(delete(s, e'), e) **fi**;

has(insert(s, e), e') = **if** e=e' **then** "true"

else has(s, e') **fi**;

size(insert(s, e)) = size(delete(s, e)) + 1

syn: for all i, j:

empty.ex → insert[i].ent ∧

empty.ex → delete[i].ent ∧

empty.ex → has[i].ent ∧

empty.ex → size[i].ent ∧

(e = e') ∧ (insert[i](e).ex → insert[j](e').ent ∨

insert[j](e').ex → insert[i](e).ent) ∧

(e = e') ∧ (insert[i](e).ex → delete[j](e').ent ∨

delete[j](e').ex → insert[i](e).ent) ∧

(e = e') ∧ (insert[i](e).ex → has[j](e').ent ∨

has[j](e').ex → insert[i](e).ent) ∧

(e = e') ∧ (delete[i](e).ex → delete[j](e').ent ∨

delete[j](e').ex → delete[i](e).ent) ∧

$$(e = e') \wedge (\text{delete}[i](e).ex \rightarrow \text{has}[j](e').ent \vee$$
$$\text{has}[j](e').ex \rightarrow \text{delete}[i](e).ent) \wedge$$
$$(\text{size}[i].ex \rightarrow \text{insert}[j].ent \vee$$
$$\text{insert}[j].ex \rightarrow \text{size}[i].ent) \wedge$$
$$(\text{size}[i].ex \rightarrow \text{delete}[j].ent \vee$$
$$\text{delete}[j].ex \rightarrow \text{size}[i].ent)$$

end set;

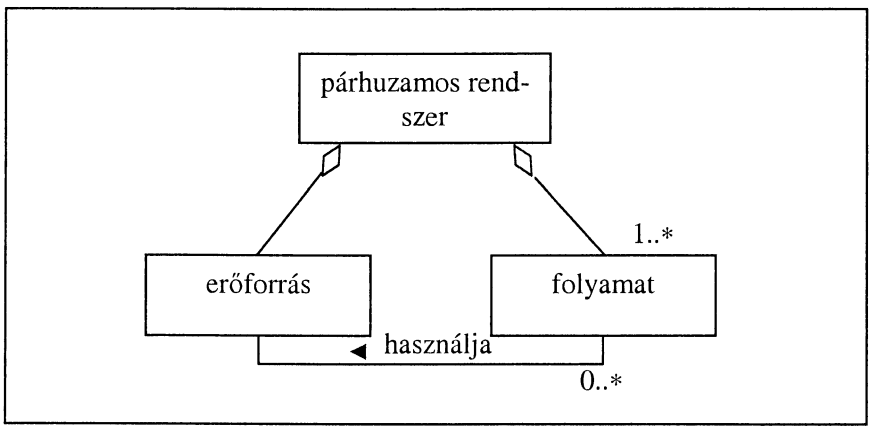
11. A SZINKRONIZÁCIÓS SPECIFIKÁCIÓ ELEMZÉSE

11.1. A HOLTPONTMENTESSÉG VIZSGÁLATA

Adva van a d_a = (A, F, E, S) az S szinkronizációs specifikációval kiegészített absztraktadattípus-specifikáció. Legyen f(a)∈F, az adattípus egyik művelete. Jelölje $S(a)_{f(a).ent}$ az S szinkronizációs specifikáció kiértékelésének az értékét az f(a) művelet enter fázisának a végrehajtási időpontjában, és legyen P egy olyan folyamat, amely ezt a műveletet igénybe veszi. Azt mondjuk, hogy a P folyamat az f(a) művelet végrehajtásának megkezdésekor blokkolt állapotba került, ha

$$S(a)_{f(a).ent} = \text{"false"}.$$

Legyen adott az (E, p) párhuzamos rendszer, amelynek osztálydiagramját a következő ábra írja le:



Az erőforrás egy $d_a = (A,F, E, S)$ specifikációval meghatározott adattípus objektuma.

Azt mondjuk, hogy az adattípus szinkronizációs specifikációja holtpontmentes, ha nem állhat elő az a helyzet, hogy

$$(\forall f \in F)(S(a)_{f(a).ent} = \text{"false"}).$$

EGY ELÉGSÉGES FELTÉTEL A SZINKRONIZÁCIÓ HOLTPONTMENTESSÉGÉNEK ELDÖNTÉSÉRE

1. Hozzuk az S(a) szinkronizációs specifikációt diszjunktív normál formára.

$$S(a) = S_1(a) \vee S_2(a) \vee \ldots \vee S_n(a).$$

2. Minden $S_j(a)$-hoz szerkesszük meg az elsőbbségi gráfot a következőképpen.
   - Legyen a gráf egy irányított gráf.
   - A gráf csomópontjait az $f[i](a).\chi$ műveleti fázisok alkotják.
   - A gráf csomópontjait minden esetben kössük össze élekkel, ha az $f[i].\chi \rightarrow g[j].\rho$ elsőbbségi reláció fennáll.
   - Ha az átmenet feltételhez kötött, $(R(a, b) \wedge (f[i].\chi \rightarrow g[j].\rho))$, az élhez rendeljük hozzá az R(a, b) feltételt.

MEGJEGYZÉS Az axiómák alapján a gráf részét képezi minden $f \in F$ esetén a következő részlet:

$$f[i].req \rightarrow f[i].ent \rightarrow f[i].ex$$
$$\downarrow k > 0$$
$$f[i + k].req \rightarrow f[i + k].ent \rightarrow f[i + k].ex$$

11.1.1. ÁLLÍTÁS Ha mindegyik $S_j(a)$-hoz megszerkesztett elsőbbségi gráf körmentes, akkor az S(A) specifikáció holtpontmentes.

BIZONYÍTÁS A bizonyítást indirekt módon végezhetjük el. Tegyük fel, hogy a rendszer holtpontra jutott. Akkor valamelyik f[i].ent esetén végtelen várakozás következik be. Ez viszont csak úgy lehet, hogy

$$f[i].ent \rightarrow g[j].\chi \rightarrow \ldots \rightarrow h[k].\rho \rightarrow f[i].ent,$$

azaz valamelyik elsőbbségi gráfban kör van.

Ezután tegyük fel, hogy minden elsőbbségi gráf körmentes és a rendszer holtpontra jutott. Ekkor

$$\text{minden elsőbbségi gráf körmentes} \Rightarrow$$

$$\text{a rendszer holtpontra jutott} \Rightarrow$$

$$\text{valamelyik elsőbbségi gráfban kör van,}$$

ami ellentmondás.

11.1.1. PÉLDA Adva van a kölcsönös kizárás szinkronizációs specifikációja:

$$S(a): (\forall i, j)(p[i].ex \rightarrow p[i + 1].ent \wedge$$

$$q[j].ex \rightarrow q[j + 1].ent \wedge$$

$$(p[i].ex \rightarrow q[j].ent \vee q[j].ex \rightarrow p[i].ent)).$$

Készítsük el az elsőbbségi gráfot és elemezzük azt.

MEGOLDÁS Adjuk meg a diszjunktív normál formát:

$$S_1(a): p[i].ex \rightarrow p[i + 1].ent \wedge$$

$$q[j].ex \rightarrow q[j + 1].ent \wedge$$

$$p[i].ex \rightarrow q[j].ent.$$

$$S_2(a): p[i].ex \rightarrow p[i + 1].ent \wedge$$

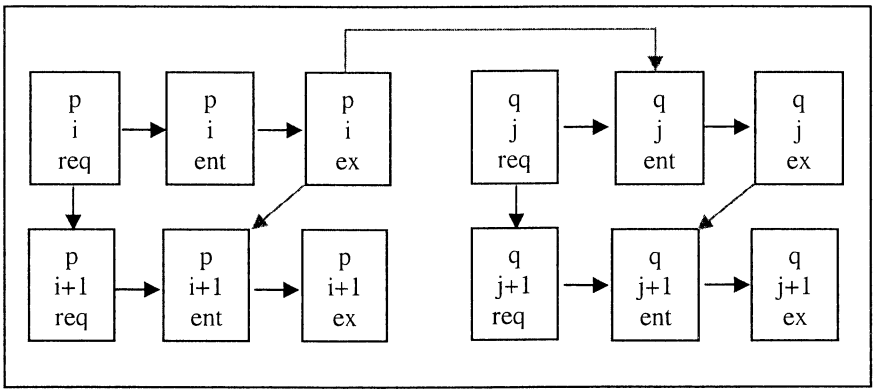
$$q[j].ex \rightarrow q[j + 1].ent \wedge$$

$$q[j].ex \rightarrow p[i].ent.$$

Két gráfot kell elkészíteni.

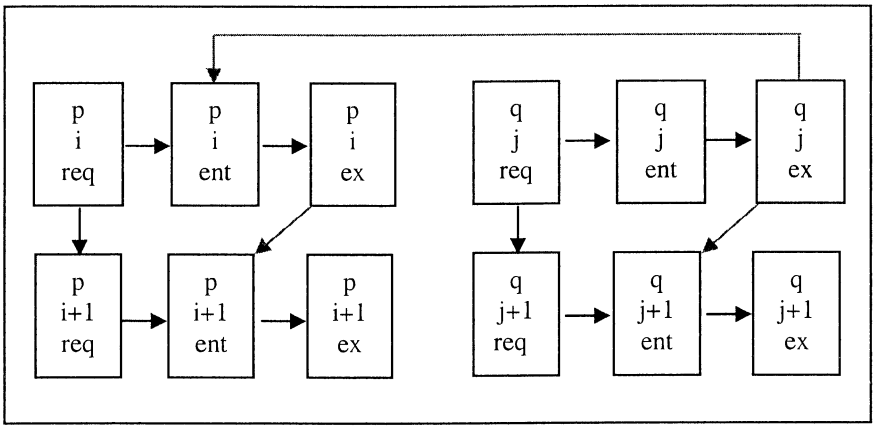
Az $S_1(a)$-hoz tartozó gráf:

$$p[i].ex \rightarrow p[i + 1].ent \wedge q[j].ex \rightarrow q[j + 1].ent \wedge p[i].ex \rightarrow q[j].ent.$$



Az S_2(a)-hoz tartozó gráf.

$p[i].ex \rightarrow p[i + 1].ent \wedge q[j].ex \rightarrow q[j + 1].ent \wedge q[j].ex \rightarrow p[i].ent.$



Az ábrák alapján nyilvánvaló, hogy a fenti két gráf körmentes.

11.1.2. PÉLDA Tekintsük a korlátos verem szinkronizációs specifikációját:

$$S(s): (\forall i, j)(push[i].ex \rightarrow pop[i].ent \wedge$$
$$pop[i].ex \rightarrow push[i + n].ent \wedge$$
$$push[i].ex \rightarrow push[i + 1].ent \wedge$$

pop[i].ex→pop[i + 1].ent ∧

(pop[i].ex→push[j].ent ∨ push[j].ex→pop[i].ent));

Itt a length-re és a top-ra vonatkozó megkötéseket elhagytuk. Készítsük el az elsőbbségi gráfot és elemezzük azt.

MEGOLDÁS A diszjunktív normál forma:

$$S(s) = S_1(s) \vee S_2(s);$$

ahol $S_1(s)$: $(\forall i, j)$(push[i].ex → pop[i].ent ∧

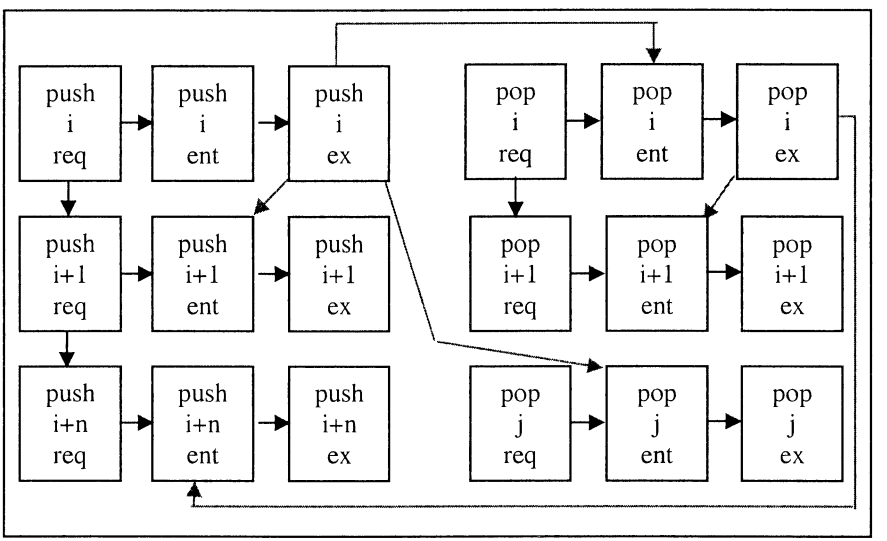
pop[i].ex → push[i + n].ent ∧

push[i].ex → push[i + 1].ent ∧

pop[i].ex → pop[i + 1].ent ∧

push[i].ex → pop[j].ent.

A diszjunktív normál forma első tagjához tartozó elsőbbségi gráf:



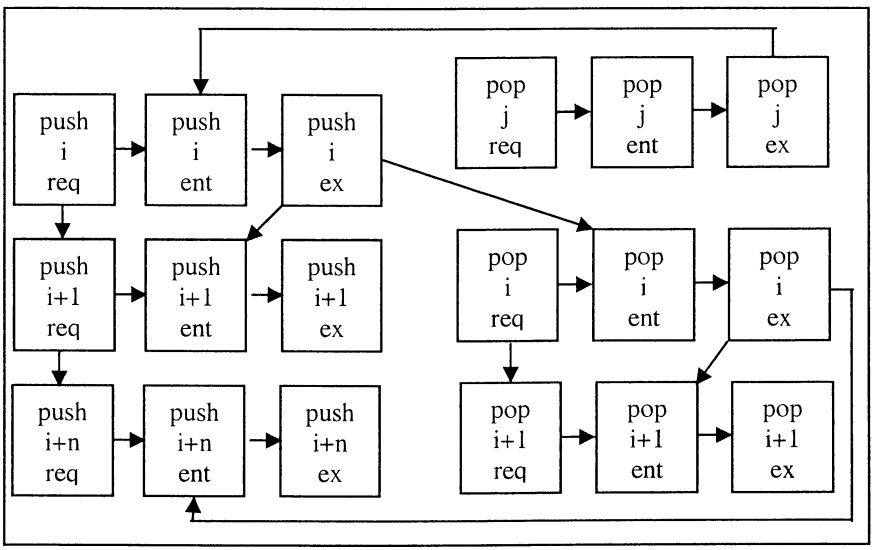
$S_2(s)$: $(\forall i, j)$(push[i].ex → pop[i].ent ∧

pop[i].ex → push[i + n].ent ∧

push[i].ex → push[i + 1].ent ∧

pop[i].ex → pop[i + 1].ent ∧

pop[j].ex → push[i].ent).



A fenti két gráf körmentessége itt is nyilvánvaló.

Az eddigi feladatokban nem volt prioritás.

11.1.3. PÉLDA Vizsgáljuk meg az adatbázis holtpontmentességét.

MEGOLDÁS S(d): (∀i, j, k)((writer[i].ex → reader[j].ent ∨

reader[j].ex → writer[i].ent) ∧

(writer[i].req → read[k].ent ⇒

writer[i].ex → reader[k].ent))

Hozzuk először S(d)-t diszjunktív normál formára. Az univerzális kvantort elhagyva az eredmény a következő:

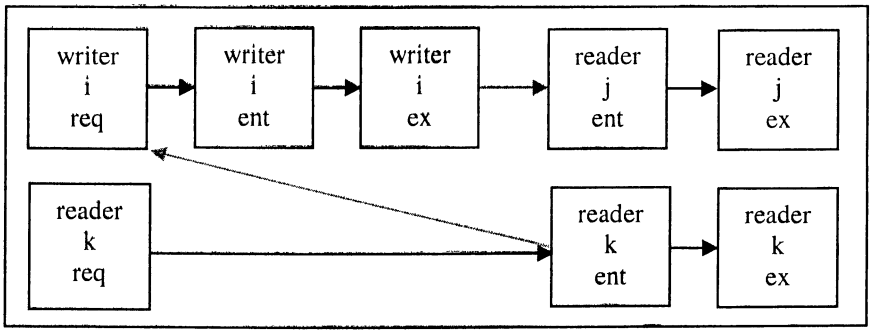
S(d) = ((writer[i].ex → reader[j].ent) ∧ ~(writer[i].req → read[k].ent)) ∨

((writer[i].ex → reader[j].ent) ∧ (writer[i].ex → reader[k].ent))∨

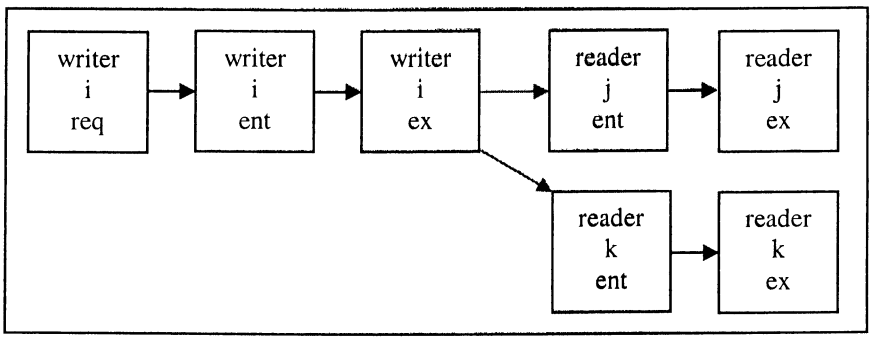
((reader[j].ex → writer[i].ent) ∧ ~(writer[i].req → read[k].ent))∨

((reader[j].ex → writer[i].ent) ∧ (writer[i].ex → reader[k].ent))

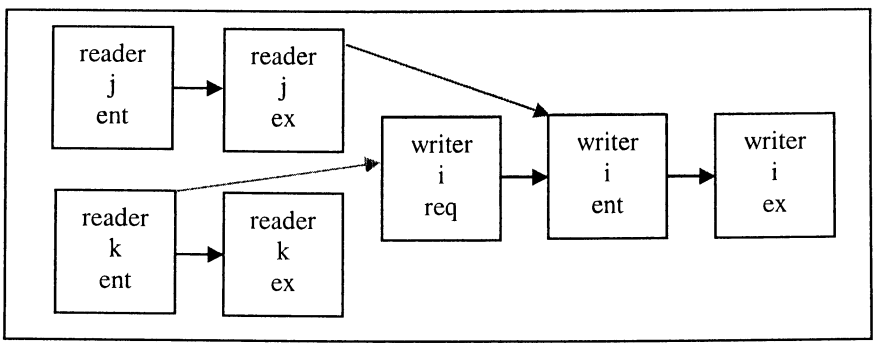
S_1(d): (writer[i].ex→reader[j].ent) ∧ ~(writer[i].req → read[k].ent)



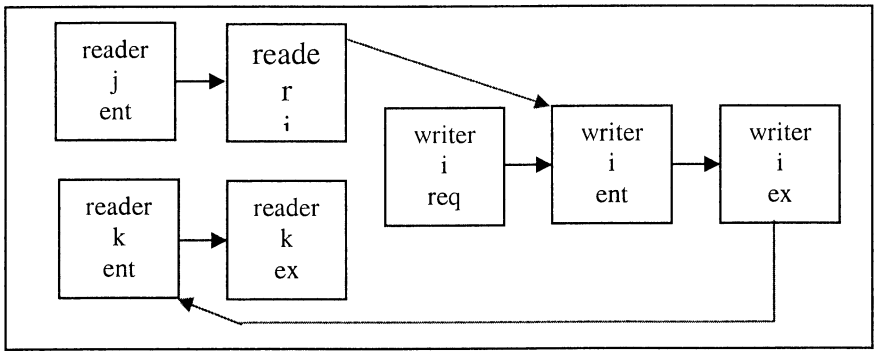
S_2(d): (writer[i].ex → reader[j].ent) ∧ (writer[i].ex → reader[k].ent)



S_3(d): (reader[j].ex → writer[i].ent) ∧ ~(writer[i].req → read[k].ent)



$S_4(d)$: (reader[j].ex → writer[i].ent) ∧ (writer[i].ex → reader[k].ent)



Ennél a feladatnál is körmentes gráfokat kaptunk.

11.2. A KONZISZTENCIA VIZSGÁLATA

Az objektumot párhuzamos környezetben két védőburok veszi körül. Az ezeken a védőburkokon való áthaladás feltételeinek egymással összhangban kell lenniük. Ez azt jelenti, hogy ha a külső feltétel teljesülése alapján a folyamat engedélyt kap arra, hogy az objektumon az adott műveletet elvégezheti, akkor ezt a belső feltétel alapján is elvárjuk.

DEFINÍCIÓ Legyen d = (A,F, E, S) típusspecifikáció, ahol a műveletek szemantikája elő- és utófeltételes formában van megadva.

{"true"} a = f_0;

{$pref_i$(a)} a' = f_i(a);

...

Legyen adott továbbá az S(a) szinkronizációs specifikáció elsőbbségi relációkkal. Tegyük fel, hogy

$(\forall i, j)(f_0.ex \rightarrow f_i[j].ent)$.

Azt mondjuk, hogy az adattípus specifikációja konzisztens, ha

$$(\forall i, j)(S(a)_{f_i[j](a).ent} \Rightarrow pref_i(a),$$

ahol $S(a)_{f_i[j](a).ent}$ az S(a) szinkronizációs specifikáció értékét az $f_i[j](a)$ művelet ent fázisának az időpontjában jelenti.

11.2.1. PÉLDA Bizonyítsuk be a kölcsönös kizárás specifikációjának konzisztenciáját.

BIZONYÍTÁS Legyen

$$me = (\{me, elem\},$$
$$\{create: \rightarrow me, p: me\ elem \rightarrow me, q: me \rightarrow elem\},$$
$$\{q(create) = \perp, q(p(m, e)) = e\}).$$

Legyen továbbá

$$S(a): (\forall i, j)(create.ex \rightarrow p[i].ent \wedge$$
$$p[i].ex \rightarrow p[i + 1].ent \wedge$$
$$p[1].ex \rightarrow q[j].ent \wedge$$
$$q[j].ex \rightarrow q[j + 1].ent \wedge$$
$$(p[i].ex \rightarrow q[j].ent \vee q[j].ex \rightarrow p[i].ent)).$$

Az axiómák átírásával kapjuk, hogy $\{p' = p(m, e)\}\ q' = q(p')$. Az S(a) alapján viszont nyilvánvaló, hogy ha S(a) = "true", akkor a p műveletnek a q műveletet meg kell előznie.

11.2.2. PÉLDA Bizonyítsuk be a verem specifikációjának konzisztenciáját.

BIZONYÍTÁS Legyen

$$d = (\{verem, elem, nat\},$$
$$\{create: \rightarrow verem, push: verem\ elem \rightarrow verem,$$
$$pop: verem \rightarrow verem\ elem, length: verem \rightarrow nat\},$$
$$\{length(create) = 0, length(push(s, e)) = length(s) + 1\}).$$

Az egyes műveletek előfeltételei:

$$\{\text{"true"}\}\ s = \text{create};$$

$$\{0 \leq \text{length}(s) < n\}s' = \text{push}(s, e);$$

$$\{0 < \text{length}(s) \leq n\}(s', e) = \text{pop}(s);$$

Adott az S(s) szinkronizációs specifikáció:

$$(\forall i, j)(\text{push}[i].ex \rightarrow \text{pop}[i].ent \wedge$$
$$\text{pop}[i].ex \rightarrow \text{push}[i + n].ent \wedge$$
$$\text{push}[i].ex \rightarrow \text{push}[i + 1].ent \wedge$$
$$\text{pop}[i].ex \rightarrow \text{pop}[i + 1].ent \wedge$$
$$(\text{pop}[i].ex \rightarrow \text{push}[j].ent \vee \text{push}[j].ex \rightarrow \text{pop}[i].ent));$$

Tegyük fel, hogy

$$(\forall i)(\text{create}.ex \rightarrow \text{push}[i].ent \wedge \text{create}.ex \rightarrow \text{pop}[i].ent).$$

A konzisztencia bizonyítását a műveletek száma szerinti teljes indukcióval végezzük el. Legyen j az elvégzett push, k pedig a pop műveletek száma. A műveletek száma tehát i = j + k.

Alapeset: Legyen i = 0. Feltevésünk szerint ekkor s = create.

Ha i = 1, két esetet kell megvizsgálni.

a.) j = 0, k = 1. Tehát az első művelet, amelynek a szinkronizációs specifikáció szabad utat adott, pop[1] volt. Ekkor azonban

$$S(s)_{\text{pop}[1].ent} = \text{"true"} \Rightarrow \text{push}[1].ex \rightarrow \text{pop}[1].ent = \text{"true"},$$

ami ellentmondás.

b.) j = 1, k = 0. Az első művelet, amely végrehajtódott, push[1]. Ekkor $S(s)_{\text{push}[1].ent}$ = "true" nem követeli meg, hogy a push műveletnek más előzménye legyen, mint a create. Feltevésünk szerint viszont a create művelet megelőzte a push műveletet, tehát az s létezik, és length(s) = 0. Így teljesül push[1](s) előfeltétele: $0 \leq \text{length}(s) < n$.

Bebizonyítottuk tehát, hogy S(s) csak a push műveletet engedi át elsőnek, de akkor a push művelet előfeltétele is fennáll. Ezután jöhet az indukció.

Indukciós lépés: Tegyük fel, hogy minden olyan j, k műveletszám esetén, amelyre $0 \leq k \leq j < i$, a tétel állítása igaz, azaz a specifikáció konzisztens.

A $0 \leq k \leq j$ feltevésünk nyilvánvalóan helyes, hiszen

$$S(s) \Rightarrow (\forall j, k)(push[j].ex \rightarrow pop[k].ent).$$

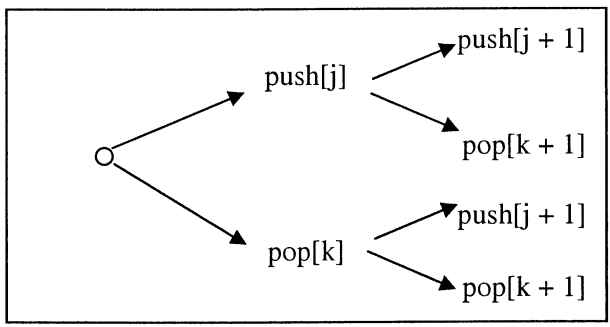
Bizonyítsuk ennek alapján azt az állítást, hogy a $0 \leq k \leq j \leq i$, esetre is igaz az állításunk, ahol $i > 1$.

Egyszerű indukcióval belátható, hogy length(s) = j - k, azaz push eggyel növeli – ez axióma –, pop pedig eggyel csökkenti a hosszat.

Két esetet kell most megvizsgálni:

- push[j] művelet volt utoljára;
- pop[k] művelet volt utoljára.

Ezek után ismét két eset vizsgálandó aszerint, hogy utána mi következik. A vizsgálati fánk tehát a következő lesz:



Vegyük tehát sorba a négy esetet.

1. push[j] művelet volt az utolsó, amelyet push[j + 1] művelet követ. Ha a push[j + 1] műveletet S(s) átengedi, akkor

$$S(s)_{push[j+1].ent} \Rightarrow pop[k].ex \rightarrow push[k + n].ent$$

igaz, ezért $j + 1 \leq k + n$, azaz $j - k = length(s) < n$.

Tehát teljesül push[i + 1] előfeltétele.

2. push[j] művelet volt az utolsó, amelyet pop[k + 1] művelet követ. Ha a pop[k + 1] műveletet S(s) átengedi, akkor

$$S(s)_{pop[k+1].ent} \Rightarrow push[j].ex \rightarrow pop[j].ent$$

igaz, ezért $k + 1 \leq j$, azaz $0 < j - k = length(s)$.

Tehát teljesül pop[k + 1] előfeltétele.

Ezután hasonlóképpen kapjuk a következő két esetet is.

3. pop[k] művelet volt az utolsó, amelyet push[j + 1] művelet követ. Ha a push[j + 1] műveletet S(s) átengedi, akkor

$$S(s)_{push[j+1].ent} \Rightarrow pop[k].ex \rightarrow push[k + n].ent$$

igaz, ezért $j + 1 \leq k + n$, azaz $j - k = length(s) < n$.

Tehát teljesül push[i + 1] előfeltétele.

4. pop[k] művelet volt az utolsó, amelyet pop[k + 1] művelet követ. Ha a pop[k + 1] műveletet S(s) átengedi, akkor

$$S(s)_{pop[k+1].ent} \Rightarrow push[j].ex \rightarrow pop[j].ent$$

igaz, ezért $k + 1 \leq j$, azaz $0 < j - k = length(s)$.

Tehát teljesül pop[k + 1] előfeltétele.

12. EGY MÓDSZER A SZINKRONIZÁCIÓS SPECIFIKÁCIÓ KONKRÉT MEGADÁSÁRA

A szinkronizációs specifikációs módszertől elvárható tulajdonságok:

- minimális legyen a mechanizmus;
- legyen alávethető öröklődésnek;
- legyenek bizonyíthatók a szinkronizációs specifikáció tulajdonságai;
- legyen alkalmazható a konkurencia széles körére.

A következőkben ilyen elveknek megfelelő konkrét specifikációs módszert kívánunk ismertetni. Ezt a módszert az adattípus megvalósítása során fogjuk alkalmazni. Ha az osztály morfizmusdiagramjára gondolunk, akkor arról van szó, hogy a specifikációs módszerünk a konkrét szinkronizációs specifikáció megadására szolgál a törzs (body) részben.

Az állapotdiagramban egy új belépés feltételét általában az állapotok invariánsaiból képezett logikai kifejezésekkel fejeztük ki. Például

$$\textbf{in}\ \langle \text{állapot}_1 \rangle \wedge \sim \textbf{in}\ \langle \text{állapot}_2 \rangle$$

Ha tehát nyilvántartjuk az állapotokba történő belépések ill. kilépések számát, akkor az ilyen logikai kifejezések ezekkel a számokkal képzett relációkkal fejezhetők ki.

Például

$$\textbf{in}\ \langle \text{állapot}_1 \rangle \wedge \sim \textbf{in}\ \langle \text{állapot}_2 \rangle \equiv$$
$$(\text{belépések száma}_1 - \text{kilépések száma}_1 > 0) \wedge$$
$$(\text{belépések száma}_2 - \text{kilépések száma}_2 = 0).$$

Egy állapotba való belépés ill. kilépés egy-egy műveleti fázis végrehajtását jelenti. A probléma megoldásához tehát a műveletek fázisainak számát kell nyilvántartanunk. Ez az alapja annak a módszernek, melyet a következőkben ismertetünk. Ez az állapotokban való tartózkodás nyilvántartásának egy módszere.

12.1. AZ ÁLLAPOTTÉR MÓDSZERE

- Jelölje a műveleti fázisok számát egy adott p(x) műveletre vonatkozóan

  count(p(x).req);

  count(p(x).ent);

  count(p(x).ex).

- A szinkronizáció állapottere a fenti számlálók halmazával definiálható:

$$A = \{count(p_i(x).\chi) \mid i = 1, 2, \ldots, n \wedge \chi \in \{req, ent, ex\}\}.$$

- Az állapotteret minden műveleti fázis végrehajtásakor módosítani kell. Ez a módosítás a következőt jelenti:

$$A' = A / \{count(p(x).\chi)\} \cup \{count(p(x).\chi) + 1\}.$$

  Jelölje ezt a műveletet a következőkben:

$$A' = m(A, count(p(x).\chi)).$$

- Egy új állapotba való belépéskor ki kell értékelnünk az állapotteret, hogy eldönthessük, vajon a szinkronizáció követelményével összeegyeztethető-e a belépésünk. A kiértékelés azt jelenti, hogy logikai kifejezéseket képezünk az állapottér elemei és a nem negatív egész számok között az aritmetikai műveletek közötti relációk segítségével.

  Például:

$$0 \leq count(p.ent) - count(q.ex) < n;$$

$$(count(p(x).req) > count(q(y).ent) \wedge (x = y);$$

- Az ilyen logikai kifejezést egy adott állapotba való belépés kapufeltételének nevezzük. Jelölje ezt a kifejezést p(x) művelet esetén k(p(x).ent).
- A művelet fázisainak a végrehajtásainak atomi utasításoknak kell lenniük.

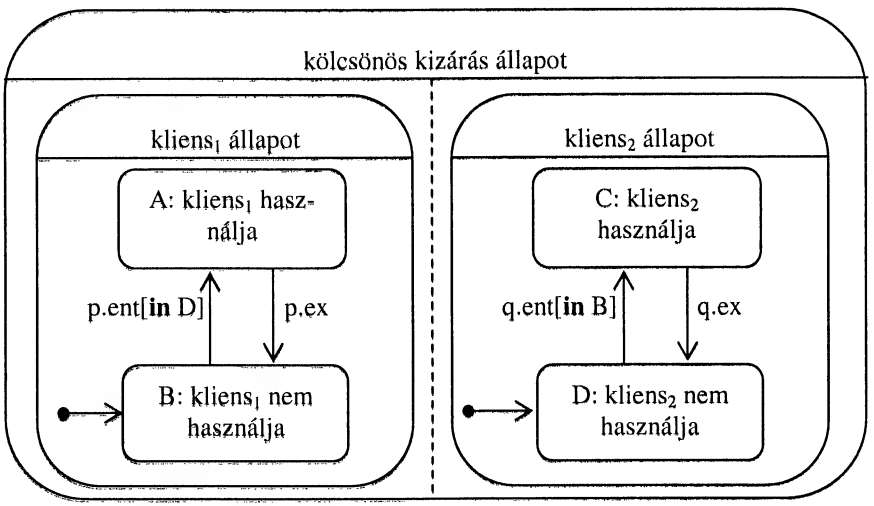
  p(x).req: **await** "true" **then** A ← m(a, count(p(x).req)) **ta**;

  p(x).ent: **await** k(p(x).ent) **then** A ← m(a, count(p(x).ent)) **ta**;

  p(x).ex: **await** "true" **then** A ← m(a, count(p(x).ex)) **ta**;
- Ezeket az atomi utasításokat végül például a korábban megismert módszer szerint P(s), V(s) szemaforutasításokkal valósíthatjuk meg.

12.1.1. PÉLDA Készítsük el a kölcsönös kizárás típusosztályának specifikációját párhuzamos környezet számára.

MEGOLDÁS Adott tehát a kölcsönös kizárás állapotdiagramja.



Az állapotdiagram alapján általában könnyen megadhatjuk a kapufeltételeket a most definiált specifikációs nyelvünkön. Esetünkben ezek a következők:

in D: count(create) > 0 ∧ count(q.ent) = count(q.ex);

in B: count(create) > 0 ∧ count(p.ent) = count(p.ex);

Ezt felhasználva írjuk fel a kölcsönös kizárás típusosztályának teljes specifikációját:

```
me(elem) is a class specification =
    parameters =
        sorts: elem
        oprs: ⊥: → elem
    exports =
        class sorts: me
            oprs: create: → me
                  p: me elem → me
                  q: me → elem
            eqns: m∈ me; e∈ elem;
                  q(create) = ⊥
                  q(p(m, e)) = e
            syn: for all i, j:
                  create → p[i].ent ∧
                  p[i].ex → p[i + 1].ent ∧
                  q[j].ex → q[j + 1].ent ∧
                  (p[i].ex → q[j].ent ∨ q[j].ex → p[i].ent)
    imports =
    body =
        eqns = x∈ me, e∈ elem;
        syn =
            procedure create begin
              await "true" then A ← empty;
                    A ← m(A, count(create.req)) ta;
              await "true" then A ← m(A, count(create.ent)) ta;
```

```
        x ← create;
        await "true" then A ← m(A,count(create.ex)) ta;
                                    end create;
      procedure p(x, e) begin
        await "true" then A ← m(A, count(p.req)) ta;
        await create.ex > 0 ∧ count(q.ent) = count(q.ex)
              then A ← m(A, count(p.ent)) ta;
        x ← p(x, e);
        await "true" then A ← m(A, count(p.ex)) ta;
                                    end p;
      procedure q(x) begin
        await "true" then A ← m(A, count(q.req)) ta;
        await create.ex > 0 ∧ count(q.ent) = count(q.ex)
              then A ← m(A, count(q.ent)) ta;
        e ← q(x);
        await "true" then A ← m(A, count(q.ex)) ta;
                                    end q;
end me;
```

12.1.2. PÉLDA Készítsük el a prioritásos kölcsönös kizárás típusosztályának kapufeltételeit párhuzamos környezet számára.

MEGOLDÁS Adott tehát a prioritásos kölcsönös kizárás állapotdiagramja. Az állapotok jelölései a következők:

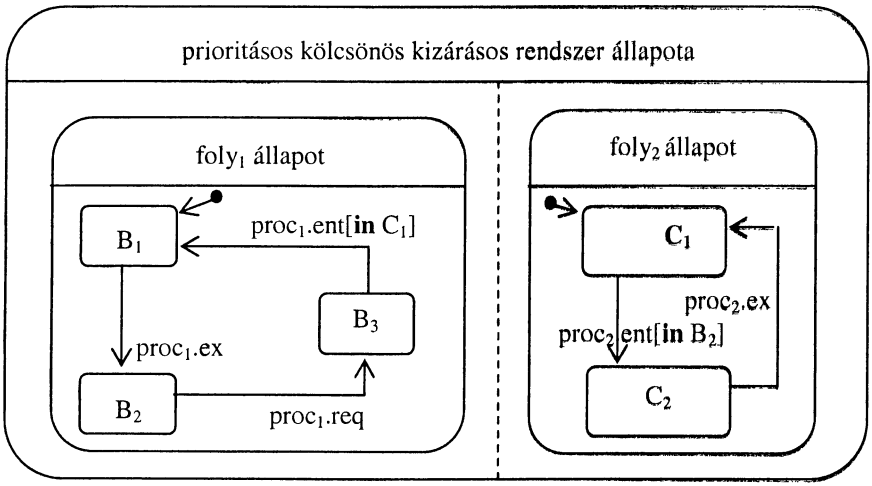
B_1: folyamat$_1$ használja az erőforrást;

B_2: folyamat$_1$ nem használja az erőforrást;

B_3: folyamat$_1$ igényli az erőforrást;

C_1: folyamat$_1$ nem használja az erőforrást;

C_2: folyamat$_1$ használja az erőforrást;



A kapufeltételek a következők:

in C_1: count(create) > 0 ∧

count($proc_2$.ent) = count(proc2.ex);

in B_2: count(create) > 0 ∧

count($proc_1$.req) = count($proc_1$.ent) ∧

count($proc_1$.ent) = count($proc_1$.ex);

12.1.3. PÉLDA Készítsük el az adatbázis adattípusának a specifikációját párhuzamos környezet számára.

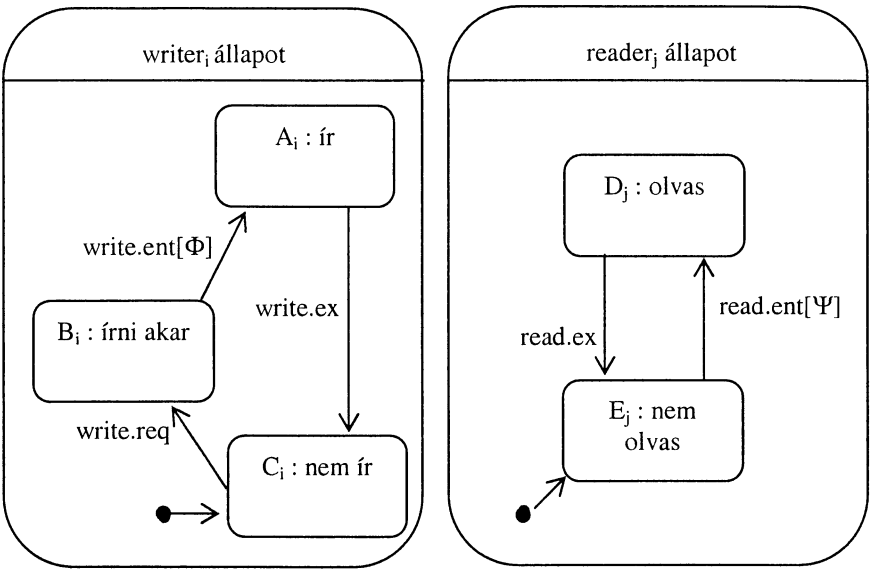
MEGOLDÁS Adva vannak a writer$_i$, reader$_j$, (i, j∈{1, ..., n} folyamatok állapotdiagramjai, ahol

Φ: (~∃j)(**in** E_j) ∧ (~∃i)(**in** A_i): count(read.ent) = count(read.ex) ∧

count(write.ent) = count(writer.ex)

Ψ: (∀i)(in A_i): count(write.req) = count(write.ent) ∧

count(write.ent) = count(writer.ex)



db(elem) **is a class specification** =

parameters = **set** +

sorts: elem

oprs: ⊥: → elem

exports =

class sorts: db

oprs: create: → db

write: db elem → db

read: db → elem

eqns: d∈ me; e∈ elem;

read(create) = ⊥

read(write(d, e)) = e

syn: for all i, j:

create.ex → writer[i].ent ∧

```
                (writer[i].ex → writer[i + 1].ent ∧
                (writer[i].ex → reader[j].ent ∨
                    reader[j].ex → writer[i].ent) ∧
                (writer[i].req → read[j].ent ⇒
                    writer[i].ex → reader[j].ent)
imports =
body =
        eqns = d∈db, e∈elem;
        syn =
            procedure create begin
              await "true"
                  then A ← empty; m(A, count(create.req)) ta;
              await "true" then A ← m(A, count(create.ent)) ta;
              d ← create;
              await "true" then A ← m(A, count(create.ex)) ta;
                                    end create;

            procedure writer(d, e) begin
              await "true" then A ← m(A, count(writer.req)) ta;
              await create.ex > 0 ∧
                  count(reader.ent) = count(reader.ex) ∧
                  count(writer.ent) = count(writer.ex)
                  then A ← m(A, count(writer.ent)) ta;
              d ← writer(d, e);
              await "true" then A ← m(A, count(writer.ex)) ta;
                                        end writer;
```

```
                    procedure reader(x) begin
                      await "true" then A ← m(A, count(reader.req)) ta;
                      await create.ex > 0 ∧
                          count(write.req) = count(write.ent) ∧
                          count(writer.ent) = count(writer.ex)
                          then A ← m(A, count(reader.ent)) ta;
                      e ← reader(m);
                      await "true" then A ← m(A, count(reader.ex)) ta;
                                          end reader;
    end db;
```

A konkrét specifikáció az f_1, ..., f_n műveletek kf_1, ..., kf_n kapufeltételeinek konjunkciója:

$$S_c = \{kf_1.ent \wedge \ldots \wedge kf_n.ent\}.$$

Például az adatbázismodell esetében a konkrét szinkronizációs specifikáció a következő:

$$Sc = \{create.ex > 0 \wedge count(read.ent) = count(read.ex) \wedge count(writer.ent) = count(writer.ex),$$
$$create.ex > 0 \wedge count(write.req) = count(write.ent) \wedge count(writer.ent) = count(writer.ex)\}.$$

13. A KONKRÉT SPECIFIKÁCIÓ ELEMZÉSE

13.1. A HOLTPONTMENTESSÉG VIZSGÁLATA

13.1. TÉTEL Legyen adott $d_c = (C, G, E_c, S_c)$.

Ha S_c :$(count(g_1.req) > count(g_1.ent) \wedge$

$count(g_1.ent) = count(g_1.ex) \wedge \sim kf_1.ent) \wedge$

...

$(count(g_j.req) > count(g_j.ent) \wedge$

$count(g_j.ent) = count(g_j.ex) \wedge \sim kf_j.ent) \wedge$

...

$(count(g_n.req) > count(g_n.ent) \wedge$

$count(g_n.ent)= count(g_n.ex) \wedge \sim kf_n.ent) \equiv$ "false",

akkor a konkrét szinkronizációs specifikáció holtpontmentes.

BIZONYÍTÁS Tegyük fel, hogy a rendszer holtpontra jutott. Ekkor az erőforráson nem dolgozhat egyetlen folyamat sem. Ugyanakkor mindegyik folyamat várakozik. Egy helyen várakozhat csak, az erőforrásra bejelentett igényének a teljesítésére. Ezek az igények azonban nem teljesíthetők, mert az összes kapu zárva van, azaz a kapufeltétel értéke "false". Ha ez az eset nem állhat elő, akkor a rendszer holtpontmentes.

13.1.1. PÉLDA Az adatbázis konkrét specifikációja holtpontmentes.

BIZONYÍTÁS $count(write.req) > count(write.ent) \wedge$

count(write.ent) = count(write.ex) ∧

~(count(read.ent) = count(read.ex) ∧

count(writer.ent) = count(writer.ex)) ∧

count(read.req) > count(read.ent) ∧

count(read.ent) = count(read.ex) ∧

~(count(write.req) = count(write.ent) ∧

count(writer.ent) = count(writer.ex))

= "false",

mivel például

~(count(read.ent) = count(read.ex) **és** count(read.ent) = count(read.ex)

is szerepel a konjunkcióban.

13.2. A KONZISZTENCIA VIZSGÁLATA

Ha minden j-re, j = 1, 2, .., n esetén bebizonyítjuk, hogy

$$kg_j.ent(c) \Rightarrow preg_j(c),$$

akkor az adattípus specifikációja konzisztens.

13.2.1. PÉLDA Készítsük el a sor specifikációját párhuzamos környezet számára.

MEGOLDÁS

que(elem, n: **natbool**) **is a class specification =**

parameters = natbool + set +

sorts: elem

oprs: ⊥: → elem

exports =

class sorts: que

oprs:

init: → que

dep: que elem → que

rem: que → que elem

length: stack → natbool

eqns: q∈que; e∈elem;

first(q, e) = q

rem(create) = "undefined"

rem(dep(create, e)) = (create, e)

length(create) = 0

rem(dep(q, e)) =dep(first(rem(q)), e)

length(dep(q, e)) = length(q) + 1

length(dep(q, e)) > n ⇒ dep(q, e) = "undefined"

syn: for all i, j:

init.ex → dep[i].ent ∧

dep[i].ex → rem[i].ent ∧

rem[i].ex → dep[i + n].ent ∧

dep[i].ex → dep[i + 1].ent ∧

rem[i].ex → rem[i + 1].ent ∧

(length[j].ex → dep[i].ent ∨

dep[i].ex → length[j].ent) ∧

(length[j].ex → rem[i].ent ∨

rem[i].ex → length[j].ent)

imports =

sorts : vector

oprs : new : → vector

put: vector natbool elem → vector

access: vector natbool → elem

$shift_L$: vector → vector

eqns : v∈vector, j, k∈natbool, e∈elem;

access(new, j) = ⊥

$shift_L$(new) = new

$shift_L$(put(v, j, e)) = **if** j = 1 **then** $shift_L$(v)

else put($shift_L$(v), j - 1, e) **fi**

access(put(v, j, e), k) = **if** j = k **then** e

else access(v, k) **fi**

body =

oprs : rep: vector natbool → que

eqns : v∈vector, m∈natbool, e∈elem;

syn =

procedure init **begin**

await "true"

then A ← empty; m(A, count(init.req)) **ta**;

await "true"

then A ← m(A, count(init.ent)) **ta**;

(v, m) ←(new, 0);

await "true"

then A ← m(A, count(init.ex)) **ta**;

end init;

procedure dep(v, m, e) **begin**

await "true" **then** A ← m(A, count(dep.req)) **ta**;

```
            await init.ex > 0 ∧
                count(dep.ent) = count(dep.ex) ∧
                0 ≤ count(dep.ex) - count(rem.ent) < n
                then A ← m(A, count(dep.ent)) ta;
            (v, m) ← (put(v, m + 1, e), m + 1);
            await "true" then A ← m(A, count(dep.ex)) ta;
                                        end dep;

        procedure rem(v, m) begin
            await "true" then A ← m(A, count(rem.req)) ta;
            await init.ex > 0 ∧
                count(rem.ent) = count(rem.ex) ∧
                n ≥ count(dep.ex) - count(rem.ent) > 0
                then A ← m(A, count(rem.ent)) ta;
            (v, m, e) ← (shift_L(v), m - 1,access(v, 1));
            await "true" then A ← m(A, count(rem.ex)) ta;
                                        end rem;
    end que.
```

13.2.2. PÉLDA Bizonyítsuk be, hogy a sor szinkronizációja holtpontmentes. A length operációra vonatkozó részt most az egyszerűség kedvéért hagyjuk figyelmen kívül.

BIZONYÍTÁS

```
count(dep.req) > count(dep.ent) ∧
~(init.ex > 0 ∧ count(dep.ent) = count(dep.ex) ∧
0 ≤ count(dep.ex) - count(rem.ent) < n) ∧
count(rem.req) > count(rem.ent) ∧
```

$$\sim(\text{init.ex} > 0 \wedge \text{count(rem.ent)} = \text{count(rem.ex)} \wedge$$
$$n \geq \text{count(dep.ex)} - \text{count(rem.ent)} > 0) \equiv$$
$$\text{"false"}$$

Itt most

$$\text{count(dep.ex)} - \text{count(rem.ent)} \geq n \text{ és } \text{count(dep.ex)} - \text{count(rem.ent)} \leq 0$$

is szerepel a konjunkcióban. Ennek alapján a konjunkció értéke "false".

13.2.3. PÉLDA Bizonyítsuk be, hogy a sor specifikációja konzisztens.

BIZONYÍTÁS Vizsgáljuk a dep esetét.

A length(create) = 0; length(dep(q, e)) > n $\Rightarrow$ dep(q, e) = "undefined" axiómák alapján kapjuk, hogy

$$\{0 \leq \text{length}(q) < n\}\ q' = \text{dep}(q, e).$$

Másrészt

$$\text{length}_c(v, m) = \text{length}_a(\varphi(v, m)),$$

ezért írhatjuk, hogy

$$\{0 \leq \text{length}(v, m) < n\}\ (v', m') = \text{dep}(v, m, e).$$

A konzisztenciához tehát dep esetében azt kell bizonyítanunk, hogy

$$\text{init.ex} > 0 \wedge \text{count(dep.ent)} = \text{count(dep.ex)} \wedge$$
$$\text{count(dep.ex)} - \text{count(rem.ent)} < n \Rightarrow 0 \leq \text{length}(v, m) < n.$$

Mivel egyszerű teljes indukcióval belátható, hogy

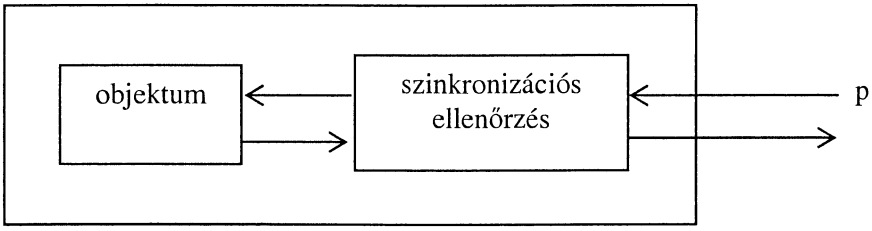
$$\text{length}(v, m) = \text{count(dep.ex)} - \text{count(rem.ent)},$$

állításunk nyilvánvalóan igaz.

Hasonlóan belátható állításunk igaz volta a rem művelet esetére is.

13.3. PÁRHUZAMOSSÁGOT VEZÉRLŐ MODUL

Az adatobjektumon egy adott műveletet csak a szinkronizáció követelményeinek teljesülése estén szabad végrehajtani. A szinkronizáció követelményeinek konkrét megfogalmazására ismertettünk egy specifikációs nyelvet. A megfogalmazott követelmények teljesülésére minden hozzáférésnél egy programnak kell ügyelnie. Úgy is fogalmazhatunk, hogy párhuzamos környezetben minden műveletnek át kell esnie egy ellenőrzési eljáráson, mielőtt az objektumon műveletet végezne. A műveletnek azonban akkor is át kell jutnia ezen az ellenőrzésen, ha befejezte a műveletet az objektumon. Ez a nyilvántartást szolgálja annak érdekében, hogy az időben változó belépési feltételt az ellenőrzés során meg lehessen határozni. Ezt az ellenőrzési mechanizmust szemlélteti a következő ábra.



Az ellenőrzési mechanizmus konkrét specifikációjára ismertünk meg az előzőekben egy nyelvet. Most először a nyelvben szereplő alapvető formulákra adunk meg könnyen megjegyezhető azonosítókat, majd egy formalizmust ismertetünk az adattípus osztályának szinkronizációt tartalmazó megfogalmazására. Ez azt a célt szolgálja, hogy rámutassunk specifikációs nyelvünk öröklődést is magába foglaló kiterjesztésének lehetőségére.

Vegyük sorra nyelvünk alapvető struktúráit és adjuk meg azok azonosítóit.

- Erőforrás használata:

  Ha p használja az erőforrást, akkor

$$count(p.ent) - count(p.ex) > 0.$$

Ha p nem használja az erőforrást, akkor

$$count(p.ent) - count(p.ex) = 0.$$

A count(p.ent) - count(p.ex) értéke tehát azt fejezi ki, hogy a közös erőforrást a p operáció lekötve tartja-e vagy sem. Célszerű erre egy kifejező rövid azonosítót használni:

Azonosító:

$$in(p) = count(p.ent) - count(p.ex)$$

- Erőforrás igénylése:

  Ha p-nek van olyan igénye, amelyik még nem teljesült, akkor

  $$count(p.req) - count(p.ent) > 0.$$

  Ha p-nek nincs olyan igénye, amelyik nem teljesült volna, akkor

  $$count(p.req) - count(p.ent) = 0.$$

  Azonosító:

  $$req(p) = count(p.req) - count(p.ent)$$

- További azonosítók, ill. rövidítések:

  $$ce(p) = count(p.ent);$$
  $$cr(p) = count(p.req);$$
  $$cx(p) = count(p.ex);$$

Az előbbiek felhasználásával tömörebben fogalmazhatjuk meg az ellenőrzést végző modul, a kontrollmodul specifikációját a következő formalizmussal.

A kontrollmodul szintaktikai formája.

control module < modul neve>;
begin p_1, ..., p_n **procedure**;
queue $part_1$ / $part_2$ / ... / $part_m$;
condition ($part_1$): < kf_1>;
condition ($part_2$): < kf_2>;
...
condition ($part_m$): < kf_m>;
end;

Itt p_1, ..., p_n a műveletek azonosítói. A műveletek között lehetnek olyanok, amelyeknek a kapufeltétele azonos. Ezeket azonos partícióba sorolhatjuk, ezeknek a partícióknak a listáját vezeti be a **queue** alapszó.
Ezután felsoroljuk a partíciókhoz tartozó kapufeltételeket.

13.3.1. PÉLDA Adjuk meg az adatbázis kontrollmodulját.

MEGOLDÁS

control module adatbázis;
begin reader, writer **procedure**;
queue reader / writer;
condition (reader): in(writer) = 0;
condition (writer): in(reader) + in(writer) = 0;
end;

13.3.2. Példa. Készítsük el a prioritásos adatbázis kontrollmodulját.

MEGOLDÁS

```
control module prioritásos adatbázis;
begin reader, writer procedure;
queue reader / writer;
condition (reader): in(writer) + req(writer) = 0;
condition (writer): in(reader) + in(writer) = 0;
end;
```

Öröklődéssel egyszerű módon származtathatók az új kontrollmodulok. Az öröklődéshez vezessük be a következő formulákat:

inherit ⟨kontrollmodulnév⟩;

Például:

inherit adatbázis;

A másik fontos dolog, hogy az átvett kapufeltételek közül azokat, amelyekre más formában van szükségünk, átdefiniálhassuk. Erre szolgál a következő formula:

redefined
  condition (partíciónév): új kapufeltétel;
rd;

13.3.3. Példa. A prioritásos adatbázis származtatása öröklődéssel.

MEGOLDÁS

```
control module prioritásos adatbázis;
begin
inherit adatbázis;
   redefined
      condition (reader): in(writer) + req(writer) = 0;
   rd;
end;
```

Írjuk fel a megismert jelöléssel a holtpontmentesség feltételét a prioritásos adatbázis estében:

$$cr(reader) > 0 \wedge cr(writer) > 0 \wedge$$
$$in(reader) = 0 \wedge in(writer) = 0 \wedge$$
$$(in(writer) + req(writer) \neq 0) \wedge$$
$$(in(reader) + in(writer) \neq 0),$$

ami természetesen egyszerűsíthető.

13.3.4. Példa. Adjuk meg a kupac kontrollmodulját.

MEGOLDÁS

```
control module kupac(n: nat);
begin hpop, htop, hpush procedure;
queue hpop, htop / hpush;
condition (hpop): in(hpush) + in(hpop) = 0 ∧ ce(hpush) - cx(htop) > 0;
condition (hpush): in(hpop) + in(hpush) = 0 ∧ cx(hpush) - cx(hpop) < n;
end;
```

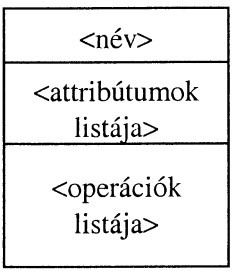
FÜGGELÉK

AZ UML JELÖLÉSRENDSZERE

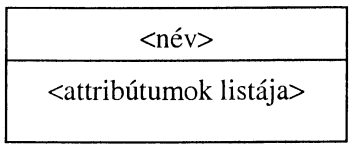
A következőkben röviden összefoglaljuk az UML modellalkotási nyelv jelölés rendszerének azokat az elemeit, amelyeket könyvünk tárgyalása során felhasználtunk. Az UML lehetővé teszi a probléma megoldására szolgáló modellek különböző szempontok szerinti megalkotását. Mi a könyvünkben csak a statikus szempontból jelentős osztálymodellt, a dinamikus szempontból fontos állapotmodellt és a végrehajtási modellt tárgyaljuk. (Szerzők ismerete szerint az utóbbi még nem képezi az UML szabvány részét). Az UML részletes megismerésére törekvők számára ajánljuk a [GRA99], [CON02], [SIK01] munkákat.

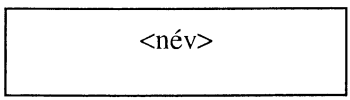
AZ OSZTÁLYDIAGRAM JELÖLÉSEI

Az osztálydiagram osztályokból és a közöttük lévő kapcsolatokból, relációkból áll. Az osztály jelölésének általános formája:

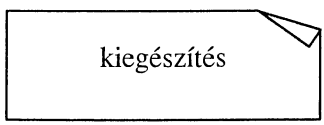


Az operáció általános jelölése: <operáció neve>( ). A diagramban azonban gyakran csak a rövidebb formákat tüntetjük fel:





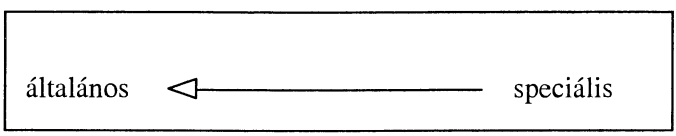
Az annotáció az UML nyelv szemantikai kiterjesztését teszi lehetővé. Jelölése:



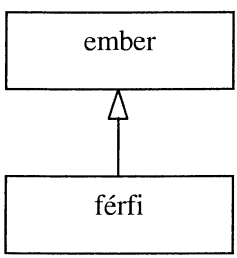
Az osztályok között fennálló relációk két csoportba oszthatók:

- osztályszintű relációk;
- osztályok objektumai szintjén fennálló relációk.

Az osztályszintű reláció az öröklődés, amelynek jele:



Például:



Az osztály objektumai szintjén fennálló relációk:

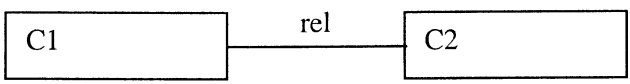
- asszociáció (társítás);
- aggregáció.

TÁRSÍTÁSI RELÁCIÓ (ASSZOCIÁCIÓ)

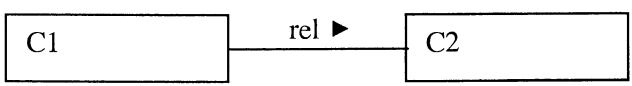
Az asszociáció két osztály közötti **absztrakt reláció**, amely kétirányú társítást fejez ki. A reláció absztrakt volta azt jelenti, hogy a reláció konkretizálása osztályok objektumainak összekapcsolásában valósul meg. Az asszociáció tehát két, vagy több osztály objektumainak valamilyen relációval történő összekapcsolása:

$$\text{Asszociáció}(C1, C2) = \{(c1, c2) \mid c1 \in C1 \wedge c2 \in C2 \wedge \text{rel}(c1, c2)\}$$

Az asszociáció jelölése:



ahol az asszociációhoz társulhat annak neve, azonosítója, amelyet itt „rel" jelöl. Az összekapcsoláshoz irány is társulhat, amely az aktív objektumtól a passzív objektum felé mutat:



Az összekapcsolásban mindkét részről több objektum is részt vehet. A j multiplicitást a reláció jele mellett az érintett osztály oldalán helyezzük el, amelynek jelentése a következő lehet:

jele	*jelentése*
j	pontosan j
0..1	0 vagy egy
i..j	i és j közötti
*	0 vagy több
i..*	i, i+1, ... valameddig

Az asszociációban résztvevő objektumnak lehet szerepe is:

- névvel azonosított szerep,
- sorrendiségi szerep, jele: {ordered}.

AZ AGGREGÁCIÓ

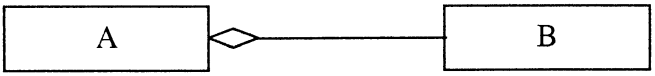
Az asszociáció az osztály objektumainak egymáshoz rendelését fejezi ki. Az egymáshoz rendelés különböző erősségű kapcsolódást jelenthet. Az asszociáció általában egymástól független osztályok társításának a kifejezésére használjuk. Az aggregáció ennél erősebb kapcsolat, amely olyan jelegű kapcsolatokat fejez ki, mint

- egész és annak részei,
- felépítmény és annak komponensei,

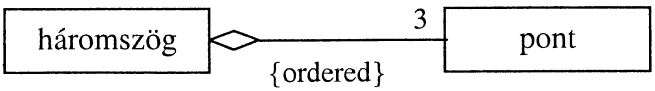
azaz

$$\text{A a B aggregációja} = \{(a, b) \mid a \in A, b \in B, b \text{ az a része}\}.$$

Az aggregáció jelölése:

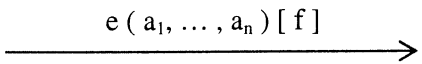


Példa:



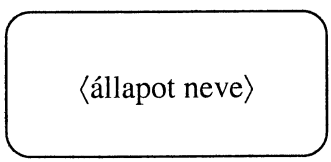
AZ ÁLLAPOTDIAGRAM JELÖLÉSEI

Az állapotdiagram egy egyszeresen összefüggő irányított gráf, amelynek csomópontjaihoz az állapotokat rendeljük, éleihez pedig az állapot átmeneteket. Állapotok közötti átmeneteket okozó eseményt az élhez rendeljük hozzá, jelölése általános formában:



ahol e az esemény azonosítója, a_1, ..., a_n paraméterek, f az esemény végrehajthatóságának feltétele.

Az állapot jelölése:

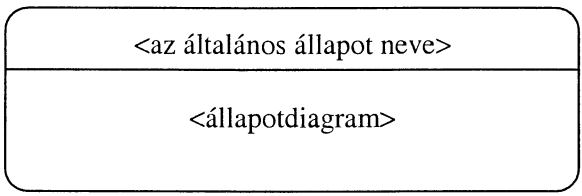


A kezdőállapot jelölése: ●

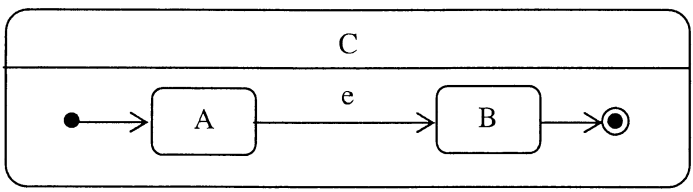
A befejező állapot jelölése: ◉

AZ ÁLLAPOTOK ÁLTALÁNOSÍTÁSA

Az általános állapot véges számú részállapot összessége. Ezek a részállapotok a az általános állapot speciális állapotai. A részállapotokhoz egy állapotdiagram tartozik, amelyben szereplő események az általános állapotot nem változtatják meg. A részállapotok örökölhetik az általános állapot tulajdonságait. Az általános állapot invariánsa a részállapotok állapot invariánsainak diszjunkciója, azaz az objektum az általános állapotban mindig valamelyik részállapotban van. Az általános állapot jelölése:



Példa:

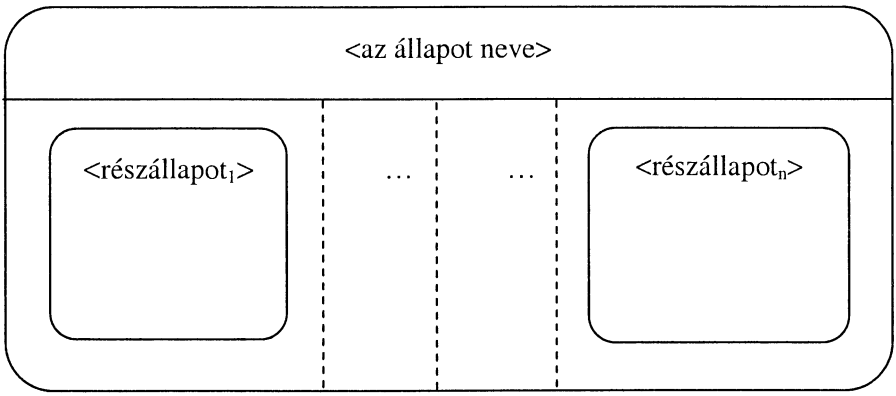


AZ ÁLLAPOTOK AGGREGÁCIÓJA

Az aggregációval létrejövő állapot egymástól független részállapotok egy véges halmaza. A részállapotok lehetnek általánosított állapotok is, amelyekhez tehát állapotdiagramok tartoznak. Minden állapotdiagramban kell lennie olyan részállapotnak, amelybe az aggregátum "entry" eseményének a hatására kerül és olyannak, amelyből való kilépéskor az "exit" esemény hatására szűnik meg. Az

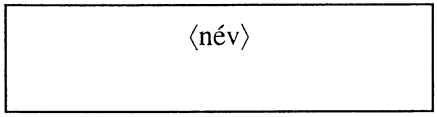
aggregációval létrejött állapot invariánsa a részállapotok invariánsainak konjunkciója, azaz az aggregációs állapot objektuma az aggregációt alkotó részállapotokban egyidejűleg létezik.

Az állapotok aggregációjának jelölése:

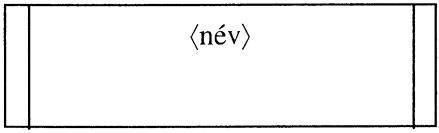


A VÉGREHAJTÁSI GRÁF JELÖLÉSEI

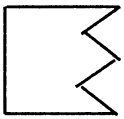
A funkció komponens jelölése:



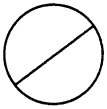
Az összetett funkció jelölése, amelyet később definiálunk.



A véletlen kiválasztás csomópontjának jele:



Az iterációs elágazás jele:



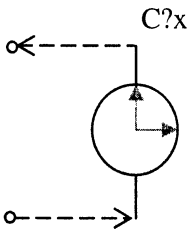
A csatornautasítások jelölései.

Szinkron csatornahívások

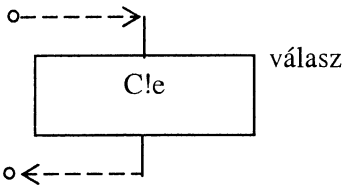
Értékadás művelete csatornautasítással, szinkron kommunikáció

Olvasás: az x változónak az e kifejezés értékét kívánjuk csatornán keresztül történő olvasással értékül adni, szinkron kommunikációt feltételezve:

Kliens esetében:

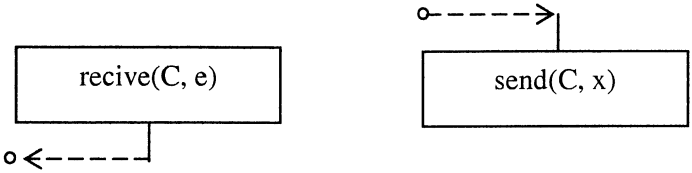


Szerver esetében:



Asszinkron csatornahívások

(nincs válasz!)



IRODALOMJEGYZÉK

[AND90] G. R. Andrews, A Method for Solving Synchronization Problems. Science of Computer Programming 13 1989/90, pp.1-21.

[AME87] P. America, Inheritance and Subtyping in a Parallel Object-oriented Language, LNCS Vol. 276, 1987, pp. 234-242.

[ANG97] Angster Erzsébet, Az objektumorientált tervezés és programozás alapjai UML Turbo Pascal C++, Angster Erzsébet, 1997.

[ATT98] P. C. Attie and E. A. Emerson, Synthesis of Concurrent Systems with Many Similar Processes ACM TOPLAS Vol. 20, No. 1, January 1998, pp. 51-115

[ATT01] P. C. Attie and E. A. Emerson, Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation ACM TOPLAS Vol. 23, No. 2, 2001, pp. 187-242

[AWA96] M. Awad et al.,Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion, Prentice Hall, Englewood Cliffs, 1996.

[BLU97] L. Blum and L. Kozma, Implementation Problems of a New Synchronization-Scheme, In: Proc. Of the Fith Symposium on Progr. Lang. and Soft. Tools, Jyväskylä, Finland, 1997, pp. 24-36.

[BOO91] G. Booch, Object Oriented Design with Applications, Redwood City, CA: Benjamin/Cummings Publishing, 1991.

[BUR93] A. Burns, G. Davies, Concurrent Programming Addison-Wesley Pub. Co. Wokingham England 1993.

[CHA89] K.M. Chandy and J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, 1989.

[CHI88] N. H. Chien and L. Varga, Algebraic Specification of an Abstract High-level Debugger, Annales Univ. Sci. Budapest, Sect. Comp. 9 1988, pp. 33-50.

[COL94] D. Coleman et al., Object-Oriented Development: The FUSION Method, Prentice Hall, Englewood Cliffs, 1994.

[CON02] C. U. Smith and L G. Williams, Performance Solutions; A Practical Guide To Creating Responsive, Scalable Software, Addison-Wesley, 2002, ISBN 0-201-72229-1.

[COO94] S. Cook and J. Daniels, Designing Object Systems: Object-Oriented Modelling with Syntropy, Prentice Hall, Englewood Cliffs, 1994.

[COA90] P. Coad and E. Yourdon, Object-Oriented Analysis, Yourdon Press/Prentice Hall, Englewood Cliffs, 1990.

[DIJ76] E.W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

[EHR85] H. Ehrig and B. Mahr, Fundamentals of Algebraic Specification 1 Equations and Initial Semantics, Springer Verlag Berlin Heidelberg, 1985, ISBN 3-540-13718-1.

[EHR90] H. Ehrig and B. Mahr, Fundamentals of Algebraic Specification 2 Module Specifications and Constraints, Springer Verlag Berlin Heidelberg, 1990, ISBN 3-540-51799-5.

[FIR93] D. Firesmith, Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach, Wiley, New York, 1993.

[FEK77] I. Fekete and L. Varga, On the formal definition of VDL-objects, Acta Cybernetica, Tom. 3, Fasc. 3, Szeged, 1977, pp. 239-248.

[FLO67] R. W. Floyd, Assigning meanings to programs, In: Proc. Sym. in Applied Math., Vol. 19, Mathematical Aspects of Computer Science (J. T. Schwartz, ed.), 1967, pp. 19-32.

[GRA94] I. Graham, Migrating to Object Technology, Addison-Wesley, Reading MA, 1994.

[GRA99] G. Booch, J. Rumbaugh and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999, ISBN 0-201-57168-4.

[GUT77] J. V. Guttag, Abstract Data Types and Development of Data Structures, Comm. ACM 20, 6, 1977, pp. 396-404.

[GUT78] J.V. Guttag, E. Horowitz and D. R. Musser, Abstract Data Types and Software Validation, Comm. ACM Vol. 21 Numb. 12, 1978, pp. 1048-1064.

[HEN94] B. Henderson-Sellers, and J. Edwards, Book Two of Object-Oriented Knowledge, Prentice Hall, Englewood Cliffs, 1994.

[HOA69] C.A.R. Hoare, An axiomatic basis of computer programming, Comm. ACM, V.12., 1969, pp. 576-583.

[HOA78] C. A. R. Hoare, Communicating Sequential Processes Comm. ACM ,Vol. 21. 1978, pp. 666-677.

[HOR93] Z. Horváth, The Weakest Precondition and the Specification of Parallel Programs, In: Proc. of the Third Symposium on Programming Languages and Software tools, Kaariku, Estonia, 1993, pp. 24-34.

[ILO91] i-Logix, The Languages of STATEMATE, Tech. Rep., i-Logix Inc., Burlington, 1991.

[ITU94] International Telecommunication Union, Criteria for the Use and Applicability of Formal Descriptions Techniques: Message Sequence Charts (MSC), Z.120 (03/93), 1994.

[JAC75] M. Jackson, Principles of Program Design, Academic Press, 1975.

[JAC83] M. Jackson, System Development, Prentice Hall, Englewood Cliffs, 1983.

[JAC92] I. Jacobson et al., Object-Oriented Software Engineering: A Use Case Driven Approach. New York, Addison-Wesley, 1992.

[KOZ81] L.Kozma, A Transformation of Strongly Correct Concurrent Programs, In: Proc. of the Third Hungarian Computer Science Conference 1981, pp. 157-170.

[KOZ82] L. Kozma ,Proving the Correctness of Implementations of Shared Data Abstractions, LNCS Vol. 137,1982, pp.227-241.

[KOZ98] L. Kozma and É. Rácz, A Specification Technique for Scheduling the Methods of Concurrent Objects, Annales Univ. Sci. Budapest, Sect. Comp. 17, 1998, pp. 253-268.

[KOZ93] L. Kozma and L. Varga, A Methodology for the Development of Shared Object Classes, In: Proc. of the ICAI'93, Eger, 1993, pp. 5-14.

[KRÖ87] F. Kröger, Temporal Logic of Programs, Springer Verlag, Berlin Heidelberg, 1987.

[KRZ97] K. R. Apt, E-R. Olderog, Verification of Sequential and Concurrent Program, Springer-Verlag, 1997, ISBN 0-387-94896-1.

[KUR96] R. Kurki-Suonio, Fundamentals of object-oriented specification and modeling of collective behaviors, Object-Oriented Behavioral Specifications (Eds. H. Kilov and W. Harvey), Kluwer, 1996, pp. 101-120.

[KUR97] R. Kurki-Suonio and Tommi Mikkonen, Liberating Object-Oriented Modelling from Programming-Level Abstractions, ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modelling Techniques Ed. by Haim Kilov & Bernhard Rumpe, Technishe Universität München, 1997, pp. 115-121.

[LAM79] A. van Lamsweerde and M. Sintzoff, Formal Derivation of Strongly Correct Concurrent Programs , Acta Informatica Vol. 12 Fasc. 1, 1979, pp. 1-31.

[LAM94] L. Lamport, The temporal logic of actions, Transactions on Programming Languages and Systems 16, 3, 1994, pp. 872-923.

[LEE90] J. van Leeuwen ed., Handbook of Theoretical Computer Science, Elsevier Science Publishers B. V., 1990.

[LIS87] B. Lisper, Synthesizing Synchronous System by Static Scheduling in Space-Time, LNCS Vol. 362, 1987.

[LOV93] T. Love, Object Lessons: Lessons Learned in Object-Oriented Development Projects, SIGS Books, Inc., New York, 1993.

[MAR81] J. Martin and C. Finkenstein, Information Engineering, Savant Institute, Carnforth, England, 1981.

[MAR95] J. Martin and J. Odell, Object-Oriented Methods: A Foundation. Prentice Hall, Englewood Cliffs, 1995.

[MAN77] Z. Manna and R. Waldinger, The automatic synthesis of recursive programs, SIGPLAN Notices, Vol. 12, No. 8, 1977, pp. 29-36.

[MAN84] Z. Manna and P. Wolper, Synthesis of Communicating Processes from Temporal Logic Specifications, ACM TOPLAS 6, 1984, pp. 68-93.

[MAW78] Z. Manna and R. Waldinger, The synthesis of structure-changing program, In: Proc. 3rd International Conference on Software Engineering, Atlanta, USA, 1978, pp. 175-178.

[MIS90] J. Misra, Specifying Concurrent Objects as Communicating Processes, Science of Computer Programming 14, 1990, pp. 159-184.

[NIE95] O. Nierstrasz and D. Tsichritzis (Edited by), Object-Oriented Software Composition, Prentice Hall, 1995.

[NAG76] Nagy Mihály és Varga László, Az inverz assembler egy általános modellje, Alkalmazott Matematikai Lapok 2, 1976, pp. 289-294.

[OWI76] S. Owicki and D. Gries, An Axiomatic Proof Technique for Parallel Programs, Acta Informatica 6, 1976, pp.319-340.

[PAG81] F.G. Pagan, Formal Specification of Programming Languages: A Panoramic Primer Prentice Hall, 1981.

[PAR94] F. Parisi-Presicce and A. Pierantonio, An Algebraic Theory of Class Specification, ACM Trans. on Softw. Eng. and Meth., Vol. 3, No. 2, 1994, pp. 166-199.

[PET83] G. L. Peterson, A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables, ACM TOPLAS, 5, 56-65, 1983.

[RÁC93] É. Rácz, Specifying a Transaction Manager Using Temporal Logic , In: Proc. of the Third Symposium on Programming Languages and Software Tools, Kaariku, Estonia, 1993, pp. 109-119.

[RAT97] Rational, Unified Modeling Language: Notation Guide, Version 1.0, Rational Software Corporation, Santa Clara, 1997.

[REE96] T. Reenskaug et al., Working with Objects: The Ooram Software Engineering Methodology, Computer Society Press, 1996.

[ROS77] D. Ross, Structured analysis (SA): A language for communicating ideas. IEEE Trans. Softw. Eng. SE-3, 1, 1977, pp. 16-34.

[ROS85] D. Ross, Applications and extensions of SADT, Computer 18, 4, 1985 pp. 25-34.

[RUM91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Object Oriented Modelling and Design, Prentice Hall, 1991.

[SEL94] B. Selic, G. Gullecson, and P. Ward, Real-Time Object-Oriented Modeling, Wiley, New York, 1994.

[SIK01] Sike Sándor és Varga László: Szoftvertechnológia és UML, ELTE Eötvös Kiadó, 2001, ISBN 963 463 477 X.

[SMU71] R.M. Smullyan ‚First Order Logic, Springer Verlag, New York 1971.

[TAI96] A. Taivalsaari, On the Notion of Inheritance, ACM Computing Surveys, Vol. 28, No. 3, 1996, pp. 438-479.

[UML99] UML in Action, Comm. of the ACM Vol. 42. No. 10, 1999.

[VAR76] L. Varga, Adatszerkezetek absztrakt szintaxisa és szemantikája, Alkalmazott Matematikai Lapok 2, 1976, pp. 41-55.

[VAR79] L. Varga, An Abstract Graph Walk Algorithm, Annales Univ. Sci. Budapest, Sect. Comp. Tomus II. 1979, pp. 63-83.

[VAR80] L. Varga, Synthesis of abstract algorithms, Acta Cybernetica, Tom. 5, Fasc. 1, Szeged 1980, pp. 59-76.

[VAR83] L. Varga, On the verification of abstract data types, Acta Cybernetica, Tom 6, Fasc. 1, Szeged, 1983, pp. 7-12.

[VAR87] L. Varga, Implementation of Abstract Data Types with Correctness Proof, Annales Univ. Sci. Budapest, Sect. Comp. 8, 1987, pp. 109-118.

[VAR88] L. Varga, Típusspecifikációk helyességének vizsgálata, Alkalmazott Matematikai Lapok 13, 1987-88, pp. 57-68.

[VÉG98] Vég Csaba és dr. Juhász István, Objektumorientált világ, IQSOFT Rt. 1998.

[WAL95] K. Waldén and J-M. Nerson, Seamless Object-oriented Software Architecture: Analysis and Design of Reliable Systems, Prentice Hall, Englewood Cliffs, 1995.

[WEG72] P. Wegner, The Vienna Definitions Language, Comput. Surveys, V. 4., 1972, pp. 5-63.

[WEG86] P. Wegner, Classification in Object-Oriented Systems, SIGPLAN Notices, Vol. 21, No. 10, 1986, pp. 173-183.

[WIE98] R. Wieringa, A Survey of Structured and Object-Oriented Software Specification Methods and Techniques, ACM Computing Surveys, Vol. 30, No. 4, 1998, pp. 459-527.

[WIL95] J. D. Williams, What Every Software Manager Must Know to Succeed with Object Technology, SIGS Books, New York, 1995.

[WIL96] C. P. Willis, Analysis of inheritance and multiple inheritance. Software Engineering Journal, July 1996, pp. 215-224.

[WIL01] Willem-Paul de Roever et al., Concurrency Verification; Introduction to Compositional and Noncompositional methods; Cambridge University Press,2001, ISBN 0 521 80608 9.

[WOL85] P. Wolper, The Tableau Method for Temporal Logic: an Overview, Logique at Anal. 28, 1985, pp. 119-136.

[YOU79] E. Yourdon and L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Prentice-Hall, Englewood Cliffs, 1979.

[YOU95] E. Yourdon et al., Mainstream Objects: An Analysis and Design Approach for Business, Prentice Hall, Englewood Cliffs, 1995.

TÁRGYMUTATÓ

A,Á

absztrakció, 6
adatabsztrakció, 13
adatbázis, 181, 186, 187, 247, 248, 249, 250, 251, 252, 254, 255, 256, 283, 284, 286, 307, 319, 331, 335, 343, 344, 345
  prioritásos, 344, 345
adattípus, 4, 14, 15, 16, 18, 19, 26, 27, 34, 35, 36, 39, 40, 45, 63, 67, 68, 69, 70, 80, 81, 91, 107, 179, 300, 314, 315, 321, 326, 336, 341
  absztrakt, 4, 13, 14, 18, 24, 26, 46, 69, 80, 89, 103
  bool, 19, 20, 22, 24, 26, 28, 29, 37, 39, 40, 43, 48, 49, 55, 56, 58, 61, 122, 126, 127, 173, 175, 186, 187, 294, 311, 312
  halmaz, 16, 20, 26, 27, 48, 70, 71, 72, 73, 75, 76, 83, 84, 92, 94, 102, 117, 203
  konkrét, 14, 16, 80, 100, 103
  kupac, 58, 345
  lista, 42, 43, 47, 54, 55, 56, 167, 173
  modnat, 30
  nat, 19, 20, 21, 22, 26, 27, 28, 29, 31, 32, 34, 39, 40, 41, 42, 48, 49, 50, 52, 58, 59, 60, 61, 62, 63, 64, 65, 71, 77, 82, 83, 90, 126, 132, 141, 142, 164, 173, 174, 175, 186, 187, 280, 281, 322, 345
  natbool, 20, 29, 30, 38, 41, 42, 48, 52, 53, 54, 55, 56, 57, 58, 61, 64, 65, 66, 77, 86, 96, 98, 163, 164, 165, 167, 168, 169, 170, 308, 311, 312, 336, 337, 338
  összetett, 27, 35, 60, 63
  sor, 39, 40, 85, 87, 88, 98, 167, 208, 209, 210, 256, 257, 279, 336, 339, 340
  táblázat, 41, 63, 77, 164, 188, 195
  vektor, 38, 64, 163, 164, 185, 196
  verem, 32, 39, 52, 53, 54, 90, 167, 168, 169, 317, 322
alaphalmaz, 27
algebra, 20, 21, 22, 25, 26, 28
  univerzális, 18, 25
állapot, 111, 122, 182, 227, 231, 232, 272, 284, 350, 351
  aggregáció, 348, 349, 350
  átmenet, 123, 199, 267
  diagram, 304, 328, 350, 351
  tér, 327
annotációs diagram, 130
atomi művelet, 184, 196, 210
atomi utasítás, 191, 195, 196, 208, 210, 211, 224, 226, 278, 279, 328
axióma, 75, 87, 92, 98, 129, 134, 167, 324

B

beütemezés, 279, 280, 281, 282, 289, 290, 291
bizonyítási rendszer megbízhatósága, 147
bizonyítási rendszer teljessége, 148
body, 46, 48, 50, 51, 52, 53, 57, 60, 63, 66, 83, 101, 127, 168, 170, 174, 176, 326, 329, 333, 338

C

ciklusszámláló, 119, 130, 149, 150

Cs

csatorna, 181, 182, 183, 292, 293, 295
csatornautasítás, 184, 293, 294
csoport, 24, 26, 246

D

Descartes-szorzat, 16, 47
determinisztikus program, 108, 110, 112, 144, 178, 198, 201, 225, 233
  diszjunkt, 200
Dijkstra, 212, 225, 238, 249, 278, 355

E,É

egyenlőség axióma, 69, 70, 73, 85
erőforrás, 179, 234, 235, 241, 245, 246, 266, 272, 283, 299, 300, 301, 303, 304, 315
  közös használatú, 179
exportfelület, 67

F

félcsoport, 24, 25
Floyd-módszer, 128, 129, 141, 144, 145, 146
folyamat, 179, 180, 191, 194, 195, 208, 209, 225, 234, 235, 241, 242, 244, 245, 247, 248, 253, 272, 273, 279, 283, 284, 291, 292, 303, 304, 305, 314, 321, 335
függvény, 6, 11, 17, 19, 21, 84, 119, 125, 152, 156, 157, 215, 301
  bijektív, 17, 23, 24
  injektív, 17, 33
  parciális, 17
  surjektív, 17
  totális, 17, 21

Gy

gyenge teljes helyesség, 229, 230, 252, 262
gyűrű, 1, 24, 300

H

Hoare-módszer, 133, 136, 140, 144, 145, 146, 148, 149, 151, 152, 153, 154, 155
holtpont, 193, 227, 228, 229, 231, 236, 295
holtpontmentesség, 231, 236, 238, 241, 262, 264, 314, 335, 345
homomorfikus feltétel, 23
homomorfizmus, 23, 24

I,Í

implementáció, 2, 3, 67, 89, 197, 248
implementáció helyessége, 89
importfelület, 46
induktív állítások, 126
  módszere, 126
információ beburkolása, 15
információ elrejtése, 15
interferencia
  -mentesség, 218, 222, 237
izomorfizmus, 23, 24

J

jól rendezett, 117, 118, 126, 130, 147
  halmaz, 118, 126, 130

K

kliens, 180, 182, 183, 184, 267, 268
kommunikáció, 179, 181, 182, 183, 292, 293, 294
  aszinkron, 181, 184, 292
  késleltetett szinkron, 181
  szinkron, 181, 183, 292
  visszahívásos aszinkron, 181
kommunikációs csatorna, 179, 292, 293
kommunikációs hálózat, 180
konfiguráció, 253
  holtpont, 227
kontrollmodul, 342, 343
konzisztencia, 321, 323, 336
kölcsönös kizárás, 178, 194, 195, 196, 197, 234, 235, 236, 237, 238, 241, 247, 248, 249, 252, 254, 255, 279, 303, 304, 305, 306, 316, 322, 328, 329
  prioritásos, 305, 330
következtetési szabály, 133, 134, 135, 136, 138, 139, 140, 146, 149, 151, 152, 155, 202, 204, 213, 214, 215, 216, 217, 228
kritikus szakasz, 194, 195, 196, 212, 235, 236, 237, 242, 243, 268, 269, 270, 271, 275
kupac, 58, 345
különbséghalmaz, 16

L

lista, 42, 43, 47, 54, 55, 56, 167, 173

M

megelőzési reláció, 302
modellalkotás, 13, 265, 266
  objektum elvű, 13
modularizáció, 7
monoid, 24, 25, 26
morfizmus, 24, 31, 32, 35, 45, 46, 82, 89
  specifikáció, 30, 34, 45, 46, 165, 166, 171, 172
művelet, 25, 28, 36, 38, 41, 68, 69, 89, 91, 194, 209, 252, 258, 300, 301, 302, 304, 305, 314, 322, 323, 324, 325, 328, 340
  konstrukciós, 27, 36, 37, 39, 40, 41, 43, 68, 70, 78, 82, 84, 85, 86, 92, 97, 104
  primitív, 204
  szelekciós, 68, 69, 70, 84
műveleti azonosító, 301, 302
műveleti fázis, 301, 302, 315, 327
műveleti forma
  kifejezés, 19

O,Ó

objektum, 1, 7, 12, 13, 15, 69, 82, 105, 106, 179, 182, 185, 299, 348, 349, 351
osztott rendszer, 2, 180, 292

Ö,Ő

őrfeltétel, 110, 210
öröklődés, 2, 16, 34, 166, 347
  specializációval történő, 165, 166
  újrafelhasználással történő, 165, 171

P

Peano-algebra, 20
primitív művelet, 204
program, 5, 6, 10, 13, 14, 90, 91, 107, 108, 109, 110, 111, 113, 114, 115, 116, 118, 122, 123, 124, 125, 129, 148, 149, 152, 155, 157, 158, 160, 161, 162, 179, 186, 187, 192, 194, 195, 196, 197, 198, 199, 200, 203, 204, 205, 212, 213, 219, 220, 225, 227, 228, 229, 231, 232, 233, 236, 237, 242, 243, 246, 249, 254, 258, 259, 263, 264, 267, 271, 272, 274, 277, 278, 286, 287, 288, 290, 293, 294, 296, 359
  determinisztikus, 108, 110, 112, 144, 178, 198, 201, 225, 233
  konkurens, 178, 179, 209, 210
  nem determinisztikus, 108, 110, 111, 113, 122, 224
  párhuzamos, 178, 184, 185, 186, 191, 194, 198, 199, 201, 207, 209, 211, 212, 213, 214, 224, 226, 227, 229, 230, 231, 232, 233, 236, 237, 262, 268, 269, 274, 275, 278, 295
  strukturált, 108, 109, 110, 112, 151, 152, 156, 158, 159, 162
  valódi, 108, 109, 115, 116, 156, 158
programfejlesztés, 7, 11
programhelyesség, 114
  parciális helyesség, 114, 115, 117, 118, 125, 126, 152, 207, 213, 261, 262
  teljes helyesség, 107, 114, 117, 119, 120, 126, 128, 129, 136, 148, 152, 204, 205, 213, 215, 228, 229, 246, 258, 260, 263
programozás, 6, 13, 180, 209, 210, 354
  objektum elvű, 1, 13, 15

Q

Q-diagram, 125, 131, 132, 148

R

reláció, 16, 17, 23, 117, 118, 119, 124, 130, 227, 301, 302, 305, 315, 347, 348, 349
  antiszimmetrikus, 17
  ekvivalens, 9, 17, 124, 144, 156, 158, 160, 162
  irreflexív, 16
  megelőzési, 302
  reflexív, 16, 17
  szimmetrikus, 17
  tranzitív, 17
reprezentáció, 2, 67, 83, 89, 98
reprezentáció elemzése, 67, 98
reprezentációs függvény, 81, 83, 84, 87, 88, 94, 95, 97, 98, 101, 102, 103, 105, 107, 128
  implicit definíciója, 83
  rekurzív definíciója, 84
reprezentációs függvények, 87, 98

S

skip, 110, 112, 129, 133, 198, 210, 212, 225, 226, 228, 297
specifikáció, 8, 9, 10, 11, 16, 22, 35, 36, 45, 48, 54, 58, 68, 70, 77, 80, 81, 86, 92, 114, 115, 116, 118, 124, 125, 126, 148, 152, 163, 165, 178, 259, 314, 315, 321, 322, 323, 324, 326, 334, 335
  absztrakt, 41, 67, 68, 78, 80, 89, 93, 99, 100, 102, 103, 104, 106, 107, 142, 301
  adekvát, 2, 9
  elő- és utófeltételes, 10, 92, 93, 107, 128, 142, 321
  kettős, 80, 100
  kitüntetett szortú, 35, 45, 46
  konkrét, 3, 80, 81, 85, 89, 93, 95, 99, 100, 103, 104, 105, 106, 107, 178, 326, 334, 335, 341
  procedurális, 7
specifikációmorfizmus, 30, 34, 45, 46, 165, 166, 171, 172
strukturális indukció, 70, 74, 75, 76, 78, 79, 85, 87

Sz

szekvencializációs szabály, 200, 201, 202
szemaforutasítás
  Dijkstra-féle, 212, 225, 278
szemantikai axióma, 70, 71
szerver, 180, 181, 183, 184, 266
szignatúra, 18, 21, 22, 31
  heterogén, 20
szignatúramorfizmus, 31, 32, 33, 34
  kiterjesztése "termekre", 33
  kiterjesztése változókra, 33
szinkronizáció, 178, 180, 212, 257, 274, 287, 299, 300, 301, 302, 303, 304, 315, 327, 341
szintaxis
  absztrakt, 360
szoftvertechnológia, 1, 5, 6, 7
szort, 18, 20, 46, 47, 167, 172
  kitüntetett, 35, 45, 46

T

term, 33, 34
típusobjektum, 70
típusobjektumok halmaza, 26
típusosztály, 4, 13, 15, 45, 47, 48, 54, 61, 67, 101, 126, 165, 171
típusöröklődés, 13, 16, 163, 165, 171
törzs, 32, 326
tranzakció, 113, 115, 123, 124, 125, 227
tranzakciós diagram, 122, 123, 124, 130, 148
tranzakciós rendszer, 124

U,Ú

univerzális algebra, 18, 25
  heterogén, 27
  homogén, 24, 26
utasítás, 133, 193, 195, 209, 210, 211, 213, 215, 224, 225, 226, 227, 228, 230, 238, 246, 279, 280, 282, 293, 294
  atomi, 191, 195, 196, 208, 210, 211, 224, 226, 278, 279, 328
  értékadó, 213, 214, 215, 229, 233, 236

Ü,Ű

üres halmaz, 16, 48
üres objektum, 69

V

változó, 47, 123, 179, 201, 204, 208, 211, 212, 235, 280, 293, 294, 295, 300, 341
végrehajtási gráf, 296, 297, 352
verem, 32, 39, 52, 53, 54, 90, 167, 168, 169, 317, 322
veremtáblázat, 63
virtuális állapot, 111, 228

Zs

zsák, 60, 61

ISBN 963 463 648 9

A könyv előállításában és műszaki szerkesztésében
közreműködött a PolgART Könyvkiadó Kft.

Nyomta és kötötte: Korrekt Nyomda
Felelős vezető: Barkó Imre

ISBN 963 463 648 9